WEST

Generate Collection Print

L65: Entry 207 of 225 File: USPT Feb 15, 1994

DOCUMENT-IDENTIFIER: US 5287511 A

TITLE: Architectures and methods for dividing processing tasks into tasks for a programmable real time signal processor and tasks for a decision making microprocessor interfacing therewith

Abstract Text (1):
Architectures and methods are provided for efficiently dividing a processing task into tasks for a programmable real time signal processor (SPROC) and tasks for a decision-making microprocessor. The SPROC is provided with a non-interrupt structure where data flow is through a multiported central memory. The SPROC is also programmed in an environment which requires nothing more than graphic entry of a block diagram of the user's design. In automatically implementing the block diagram into silicon, the SPROC programming/development environment accounts for and provides software connection and interfaces with a host microprocessor. The programming environment preferably includes: a high-level computer screen entry system which permits choosing, entry, parameterization, and connection of a plurality of functional clocks; a functional block cell library which provides source code representing the functional blocks; and a signal processor scheduler/compiler which uses the functional block cell library and the information entered into the high-level entry system to compile a program and to output source program code for a program memory and source data code for the data memory of the (SPROC), as well as a symbol table which provides a memory map which maps SPROC addresses to variable names which the microprocessor will refer to in separately compiling its program.

Brief Summary Text (10):
A.1 Functional description of The Parallel Port

Brief Summary Text (20):
A.11 Parallel Port Registers

Brief Summary Text (23):
B.1.1 The SPROCcells Function Library

Brief Summary Text (39):
C.2 Compiling SDL Files

Brief Summary Text (44):
D. The SPROC Compiler

Brief Summary Text (54):
E.9 Using the SPROC C Function Library

Brief Summary Text (60):
Appendix B--Selections from SPROCcells Function Library Source Code (pgs. 1-17)

Brief Summary Text (63):
Appendix E--SPROC Scheduler /Compiler Phantom Block Source Code (pgs. 1-22)

Brief Summary Text (68):
Appendix J--Maintest.c File Generated to Accompany FIG. 11 Example and for Compilation by Microprocessor (pgs. 1-7)

Brief Summary Text (71):
Appendix M--SPROC Scheduler /Compiler Source Code (pgs. 1-673)

Brief Summary Text (77):
Digital signal processing has evolved from being an expensive, esoteric science used primarily in military applications such as radar systems, image recognition, and the like, to a high growth technology which is used in consumer products such as digital audio and the compact disk. Single chip digital signal processors (SCDSPs) were introduced in the early 1980's to specifically address these markets. However, SCDSPs are complex to design and use, and have significant performance limitations. In particular, SCDSPs are limited to a frequency spectrum from DC to the low tens of KHz. Moreover, most SCDSPs have other development environment and hardware performance problems which stem from their Von Neuman, microprocessor origins. In an attempt to overcome these limitations, attempts have been made to use parallel processors and math coprocessors. However, these "solutions" have required considerable expertise on the part of the software engineer and have typically yielded minimal gain; particularly in the real-time environment.

Brief Summary Text (79):
Based on the above break-down of tasks it can be seen that SCDSPs are called upon to do both of what may be termed "signal processing" and "logic processing". Signal processing is typically computationally intensive, requires low latency and low parasitic overhead for real time I/O, must efficiently execute multiple asynchronous deterministic processes, and be controllable. Real time signal processors are typically controllable processors which have very large I/O bandwidths, are required to conduct many millions of computations per second, and can conduct several processing functions in parallel. In contrast to signal processing, logic processing is usually memory intensive (as opposed to computationally intensive), must efficiently handle multiple interrupts (particularly in a multiprocessor system), and acts as a controller (as opposed to being controllable). A common type of logic processor is the microprocessor which relies on extensive decision oriented software to conduct its processes. This software is typically written in a high level language such as "C". The code often contains numerous "if . . . then . . . else" like constructs which can result in highly variable execution times which are readily dealt with in nonreal time applications, but present highly problematical scheduling problems for efficient real time systems.

Brief Summary Text (84):
It is a further object of the invention to provide a graphic user interface system for a real time signal processor interfacing with a host microprocessor where the real time signal processor program is compiled separately from the program of the microprocessor but, as part of the compiling procedure provides a microprocessor-related file to the microprocessor which then translates the file and incorporates the translated file into its compilation, and thereby automatically provides for the signal processor - microprocessor interface.

Brief Summary Text (85):
Yet another object of the invention is to provide a user interface system incorporating a real time signal processor and a microprocessor which automatically share processing tasks in an efficient manner and which automatically compile and interface to accomplish the desired processing task.

Brief Summary Text (86):
In accord with the objects of the invention a development system for the microprocessor-interfacing signal processor is provided. For purposes of clarity and simplicity, the signal processor which interfaces with the microprocessor is referred to hereinafter as a SPROC (a trademark of the assignee hereof). Details of the SPROC are set forth in parent application Ser. No. 07/525,977. The development system (hereinafter referred to as SPROClab--a trademark of the assignee hereof) which is provided to permit a user to simply program and use the SPROC generally includes:

Brief Summary Text (87):
a high-level computer screen entry system (graphic user interface) which permits

choosing, entry, parameterization, and connection of a plurality of functional blocks;

Brief Summary Text (88):
a functional block library which provides source code representing the functional blocks; and

Brief Summary Text (89):
a signal processor compiler for incorporating the parameters of the functional blocks as variables into the functional block library code and for compiling the library code as well as other code which accounts for scheduling and functional block connection matters, etc., whereby the signal processor compiler outputs source program code for a program memory of the signal processor (SPROC), source data code for the data memory of the SPROC, and a symbol table which provides a memory map which maps variable names which the microprocessor will refer to in separately compiling its program to SPROC addresses.

Brief Summary Text (91):
With the signal processing and logic processing aspects of tasks being divided (with the SPROC handling the signal processing, and the microprocessor handling the logic processing), the compiling of the SPROC and the microprocessor are handled separately. In order to accomplish the separate handling while still providing the graphic entry system, at least two schemes are provided. A first scheme effectively provides graphic entry for the signal processing circuit only. If desired, in the first scheme limited graphic entry for the microprocessor can be used to provide SPROC interfaces with the microprocessor (as shown in FIG. 10). With the first scheme, the user must provide suitable code for the microprocessor separately, and the symbol table generated by the SPROClab compiler is provided together with the code hand-generated by the user for microprocessor compiling. A second scheme permits graphic entry for both the signal processing and logic processing (microprocessor) circuits, and uses any of several methods for distinguishing between the two. Among the methods for distinguishing between which portion of the circuit is intended for signal processing and which for logic processing are: user entry (e.g., defining a block as block.spr or block.mic); hierarchical block entry which is programmed to allow entry of both logic processing and signal processing blocks; and the sample rate of the block (with slow sampling rates being handled by the microprocessor). Of course, if all blocks are predefined (i.e., are contained in a library), the precoded library code divides the code into code intended for the SPROC and code intended for the microprocessor. Regardless, where graphic entry for both signal processing and logic processing is permitted, the graphic entry eventually results in separate automatic compilation for both the SPROC and the microprocessor, with the SPROClab compiler again providing the necessary symbol table for incorporation during compilation of the microprocessor code.

Brief Summary Text (92):
Additional objects and advantages of the invention will become apparent to those skilled in the art upon reference to the detailed description taken in conjunction with the provided figures.

Drawing Description Text (12):
FIG. 10 is a flow diagram of the development system of the invention where the SPROC code and microprocessor code are compiled separately.

Drawing Description Text (14):
FIG. 12 is a high level flow chart of the compiler utilized in the development system of the invention.

Detailed Description Text (4):
The SPROC 10 of the invention can function in several different modes, some of which are determined by externally set pins (not shown). In particular, the SPROC 10 has a boot mode, an operational mode, and a development mode which includes a "break" mode. In addition, the SPROC may be a master SPROC or a slave SPROC which is either coupled to a master SPROC (see FIG. 9) or a host 180 such as a microprocessor. In the boot mode (powering up), where the SPROC 10 is a master, the SPROC 10 is required to program both itself and any other slave SPROCs which might be part of

the system. To do that, upon power up, switches 192 and 194 are toggled to connect to the B (boot) nodes. With switches 192 and 194 so set, the boot ROM is coupled to a GSP 400 such as GSP 400a, and the program RAM 150 is coupled to the data RAM bus 125. As boot ROM 190 is coupled to the GSP 400a, the GSP 400a is able to read the boot code in boot ROM 190. The code is arranged to cause the GSP to seize control of the host port 800 and to load information into the SPROC from EPROM 170 via the host port 800. The information contained in EPROM 170 includes the program code for the program RAM 150 (which is sent via data RAM bus 125), configuration information for the DFM 600 and the serial, host, and access ports 700, 800, 900, and parameter information including initialization information for the data RAM 100. This information, which was compiled by the development system of the invention (as discussed in more detail hereinafter) and stored in the EPROM, causes the SPROC to perform the desired functions on data typically received via serial ports 700.

Detailed Description Text (8):
In operational mode, serial data flow into and out of the SPROC 10 is primarily through the serial ports 700, while parallel data flows through the host port 800. Serial data which is to be processed is sent into an input port 700 which is coupled to the data flow manager 600, which in turn forwards the data to appropriate locations (buffers) in the data RAM 100. In certain circumstances, described below, the DFM 600 will also write additional information to particular data RAM locations which are monitored by flag generating decoder 196. Decoder 196, in turn, causes the flags to be triggered over trigger or flag bus 198 as described in detail in previously incorporated U.S. Ser. No. 07/583,508. Other flags are triggered by pulsing hardware pins (not shown) via lines called "compute lines". The hardware pins are particularly useful in providing external timing information to the GSPs 400 and the DFM 600 of the SPROC.

Detailed Description Text (9):
Once the data has been sent to the data RAM 100, and typically after the GSPs 400 have been apprised via the flag bus 198 of the arrival of the information, the GSPs 400 can process the data. The processing of the data is conducted in accord with one or more programs stored in the multiported program RAM 150 which in turn represents the functions, topology, and parameters of a schematic diagram generated by the user of the development system. In processing the data, the GSPs 400 can read from and write to the data RAM 100. However, in order to shield the GSPs from I/O functions which would interrupt and burden the GSPs, the GSPs do not address each other directly, and do not read from or write to the DFM 600 or the input or output serial ports 700. Similarly, the GSPs do not have direct access to the host port 800 or the access port 900. Thus, in order for the processed data to be output from the SPROC 10, the processed data must be sent by the GSP 400 to the data RAM 100. The data in the data RAM is then either read by the DFM 600 and sent out serially via an output port 700, or is sent out over the host bus 165 in a parallel form via the host port 800.

Detailed Description Text (12):
Before turning to the details of each of the blocks which comprise FIG. 1, it should be appreciated that central to functioning of the SPROC is a multiported data RAM 100 and a multiported program RAM 150. As aforementioned, the RAMs may either be multiported by time division multiplexing a single access to the RAMs (as seen by the solid lines of FIG. 1) or by providing true multiported RAMs (as suggested by the dashed lines of FIG. 1). As indicated in FIG. 2, in the preferred embodiment hereof, access to the program RAM 150 by the GSPs 400 and the host port 800 and access port 900 is via time division multiplexing of a single input. Similarly, access to the data RAM 100 by the GSPs 400, the DFM 600, the host port 800, the access port 900, and the probe 1000 is also via time division multiplexing of a single input.

Detailed Description Text (13):
As seen in FIG. 2, in the preferred embodiment of the invention, there are five principle time slots of the basic 50 MHz SPROC clock 147 (shown in FIG. 1): one for each GSP: and one shared by all of the other blocks of the SPROC. Each GSP 400 is able to read from the program RAM (p-rd) once over five clock cycles, effectively providing each GSP with a 10 MHz access to the program RAM 150. In the fifth clock cycle, the host is given preferred access to either read from or write to the

program RAM. If the host does not need to read or write to the program RAM, the access port is given access. Alternatively, the host and access ports can be given 50/50 access to the fifth time slot by additional time division multiplexing.

Detailed Description Text (45):
Besides the data flow circuitry, each DFM is arranged with buffers, counters, gates, etc. to generate data RAM FIFO addresses for the incoming data. As shown in FIG. 4a, the DFM 600 has three registers 620, 622, 624, three counters 630, 632, and 634 associated with the three registers, an adder 636, a divide by two block 637, a multiplexer 638, seven logic gates 641, 642, 643, 644, 645, 646, and 647 (gates 642, 643, 645, and 647 being bus wide gates), and two delay blocks 648 and 649. The three registers are respectively: the start of FIFO register 620 which stores the start location in the data RAM for the FIFO to be addressed by the particular serial port coupled to the particular part of the DFM; the index length register 622 which stores the number of buffers which comprise the FIFO (for the FIFO of FIG. 4c, the index length register would be set at four), and the buffer length register 624 which stores the length of each buffer, i.e. the number of words that may be stored in each buffer (for the FIFO of FIG. 4c, the buffer length register would be set at eight). When a data word (twenty-four bits) is ready for sending to the data RAM for storage in a FIFO, the serial port 700a provides a ready signal which is used as a first input to AND gate 641. The second input to AND gate 641 is a data enable signal which is the time division multiplexed signal which permits the DFM to place a word on the data RAM bus. With the data enable and ready signals high, a high signal is output from the AND gate which causes driver 613 to output the data on the data RAM bus along with an address. The address is that which is computed by the twelve bit adder 636, or a prewired address, as will be described hereinafter.

Detailed Description Text (46):
When AND gate 641 provides a high output, the high output is delayed by delay blocks 648 and 649 before being input into clock counters 630 and 634. As a result, counters 630 and 634 increase their counts after an address has been output on the data RAM bus. When counter 630 increases its count, its count is added by the twelve bit adder 636 to the FIFO start location stored in register 620. If selected by multiplexer 638, the generated address will be the next address output in the address slots of the data RAM bus in conjunction with the data provided by driver 613. Thus, as data words continue to be sent by the serial port for storing in the data RAM FIFO, they are sent to incremental addresses of the data RAM, as the counter 630 increasingly sends a higher value which is being added to the FIFO start location. As is hereinafter discussed, the counter 630 continues to increase its count until a clear counter signal is received from circuitry associated with the index length register 622. When the clear counter signal is received, the counter starts counting again from zero.

Detailed Description Text (49):
As aforementioned, when an indication of a full buffer is output by bus wide XNOR gate 643, counter 632 is incremented. Counter 632 therefore tracks the number of the buffer in the FIFO that is being filled. When the number of the FIFO buffer being addressed (as determined by counter 632) is half of the FIFO length (as determined by the length stored in register 622, divided by divide by two block 637), a flag is raised by the DFM via the bus wide XNOR gate 647. The "mid buffer" flag indicates that the buffer in the FIFO being written to is halfway through the FIFO. Hence, if all previous buffers in the FIFO are still full with data, the FIFO is half full. In addition, the mid buffer flag causes the generated data input to multiplexer 611 to be changed, such that the msb of the data is a "1" instead of a zero. Thus, upon filling the buffer which causes the FIFO to be half filled, a slightly differently coded data word is placed in the data slots of the data RAM bus.

Detailed Description Text (53):
The output section of the DFM is preferably comprised of several registers and counters, logic elements including AND gates, comparators, and inverters, divide and add blocks, flip-flops, a buffer and a parallel to serial converter. Basically, the data flow through the serial output section of the DFM is simple. An address generated by the the start address register 652 is added by adder 654 to the value in the offset counter 656, and that address is output onto the address section of the data RAM bus. The data RAM receives the address information and then places the

data located at that data RAM address on the data RAM bus. That data is received by the DFM and latched and stored in buffer 694 prior to being forwarded to the serial output port 700b.

Detailed Description Text (58):
The remaining blocks of the DFM output section include a FIFO length register 680, a buffer length register 682, a sample counter 684, a divide by two block 685, comparators 686 and 687, a bus wide OR gate 689, and a set/reset block 690. The FIFO length register 682 stores the full length of the FIFO. When the value of the offset counter 656 is equal to the FIFO length stored in buffer 680, a sync pulse is generated by bus wide XNOR gate 686 which is used to synchronize the incoming data signal into an input section of a DFM with the outgoing data signal from the described output DFM. The sync pulse generated is received by the input section of the DFM (seen in FIG. 4a) as the signal enbufl, previously described. In addition the sync pulse may be used to reinitialize the DFM by clearing the offset counter 656 and reloading the registers. When the value in the offset counter 656 is equal to one-half the value of the FIFO length register 680 (as determined by divide by two block 685), comparator 687 provides a pulse to set/reset block 690 which is indicative of the fact that the address placed on the data RAM bus is the address half-way through the data RAM buffer associated with the particular DFM. When the data RAM address is the half-full address, the data being written into the data RAM buffer should not be written into the half-full address (i.e. there should never exist a situation where the address is being written to and read from at the same time). Thus, if D type flip-flop 667 provides a high signal to AND gate 670 while the DFM is running, and the output from set/reset block 690 is also, high, AND gate 698 provides a high output which sets an error flag for the DFM.

Detailed Description Text (61):
Turning to FIG. 5a, a block diagram of the serial input port 700a of the invention is seen. The basic function of the serial input port is to receive any of many forms of serial data and to convert the received serial data into parallel data synchronous with the internals of the SPROC and suitable for receipt by the DFM 600 and for transfer onto the data RAM bus 125. To accomplish the basic function, the serial input port has a logic block 710, a data accumulation register 720, and a latched buffer 730. The logic block 710 and the data register 720 are governed by seven bits of information programmed into the serial input port 700a upon configuration during boot-up of the SPROC 10. The seven bits are defined as follows:

Detailed Description Text (65):
The serial data output port 700b seen in FIG. 5b is similar to the data input port 700a in many ways, except that its function is the converse. The serial output port 700b includes a buffer 740, an parallel load shift register 750, and controlled multiplexers 760 and 770. The data to be written from the SPROC via the output port 700b is received by the buffer 740 from buffer 694 of the DFM 600. The twenty-four bits received are then loaded in parallel into the parallel load shift register 750 which functions as a parallel to serial converter. The twenty-four bits are then forwarded in a bit serial fashion via multiplexer 760 which receives the control signals dw0 and dw1, and via multiplexer 770 which receives the msb control signal to the transmit data line. Multiplexers 760 and 770 effectively transform the twenty-four bit word received by the parallel load shift register into the desired format for communication with a desired device external the SPROC. The twenty-four bits may be transformed into an eight bit word (e.g. the eight msb's), a twelve bit word, or a sixteen bit word (the eight lsb's being truncated), with either the lsb or the msb being transmitted first. A twenty-four bit word may similarly be sent lsb or msb first. Where the SPROC is communicating with another SPROC (i.e. output port 700b of one SPROC is communicating with the input port 700a of another SPROC), multiplexers 760 and 770 are preferably controlled to send a twenty-four bit word, msb first.

Detailed Description Text (67):
In slave mode (master/slave pin 801 set to slave mode), the SPROC 10 appears to other apparatus, including host microprocessors or DSPs as a RAM. Because it is desirable that the SPROC interface with as many different types processors as possible, the host port 800 is a bit parallel port and is arranged to interface with

eight, sixteen, twenty-four, and thirty-two bit microprocessors and DSPs. The mode pins 802, 804, and 806 are used to inform the host port 800 as to whether the host processor is an eight, sixteen, twenty-four bit, or thirty-two bit processor, and whether the word being sent first is the most or least significant word.

Detailed Description Text (74):
A.1 Functional description of The Parallel Port

Detailed Description Text (75):
The parallel port (PPORT0) is a 24-bit asynchronous, bidirectional port with a 16-bit (64K) address bus. The port allows for 8-,16-, or 24-bit parallel data transfers between the SPROC chip and an external controller, memory-mapped peripheral, or external memory. The port has programmable WAIT states to allow for slow memory access. A data acknowledge signal is also generated for this interface.

Detailed Description Text (80):
The following subsections describe data transfers via the parallel port for different sources and destinations. In all types of parallel port data transfers, signal values at the slave SPROC chip's mode (MODE[2:0]) and address (ADDRESS[15:0]) inputs must be stable before the chip select (CS) and read (RD), or chip select and write (WR) request goes LOW. At that time, the address is latched into the slave SPROC chip. Subsequently, after values on the data bus (DATA[23:0]) become valid, data is latched at the destination on the rising edge of the request.

Detailed Description Text (81):
To allow asynchronous communication with slow peripherals in master mode, the parallel port supports programmable WAIT states. In a preferred embodiment, a maximum of seven WAIT states are possible, where each state corresponds to one SPROC chip machine cycle, or five master clock pulses.

Detailed Description Text (82):
The parallel port also generates a handshaking signal, DTACK (data transfer acknowledge) in slave mode. This normally-HIGH signal goes LOW when the SPROC chip presents valid data in a read operation, or is ready to accept data in a write operation. DTACK is cleared when the external RD or WR strobe goes HIGH.

Detailed Description Text (83):
If enabled, a watchdog timer monitors all data transfers, and resets the parallel port if the transaction time is greater than 256 machine cycles.

Detailed Description Text (85):
A master SPROC chip initiates a read operation from a memory-mapped peripheral or external memory by reading an off-chip memory location. Prior to initiating the READ, the master SPROC chip should set up the communication mode. This includes 8-, 16-, or 24-bit data select, msb/lsb byte order, and number of WAIT states required for the peripheral. The master's internal parallel port mode register controls these options, and therefore should have been previously written to. In master mode, three bits of the parallel port mode register determine number and order of bytes transferred and are output at pins MODE[2:0]. These pins should be connected to the corresponding slave SPROC chip pins, which function as inputs in slave mode, to ensure the slave's communication mode matches the master's.

Detailed Description Text (86):
After a read cycle is initiated by the master SPROC chip, no further read or write requests to the parallel port are possible until the current read cycle has been completed. The parallel port will set up a stable address and then drive the RD strobe LOW. The strobe will remain LOW for the number of WAIT states configured in the master's parallel port mode register, and will then be driven HIGH. The data resident on the data bus will be latched into the master SPROC chip on the rising edge of the RD strobe.

Detailed Description Text (87):
If the transmission mode is 8- or 16-bit format, the read cycle will be repeated with the next extended address-output, as determined by the state of EADDRESS[1:0], until 24 bits of data have been received. The master's parallel port input register

is then updated, and the read cycle is complete. The GSP in the master that initiated the read operation must then read the contents of the parallel port input register. With the read cycle completed, the data bus I/O drivers will be reconfigured as output drivers to prevent the data bus from floating. The address bus will be driven with the last address.

Detailed Description Text (89):
A master SPROC chip initiates a write operation to a memory-mapped peripheral or external memory by writing to an off-chip memory location. Prior to initiating the WRITE, the master SPROC chip should set up the communication mode. This includes 8-, 16-, or 24-bit data select, msb/lsb byte order, and number of WAIT states required for the peripheral. The master's internal parallel port mode register controls these options, and therefore should have been previously written to. In master mode, three bits of the parallel port mode register determine number and order of bytes transferred and are output at pins MODE[2:0]. These pins should be connected to the corresponding slave SPROC chip pins, which function as inputs in this mode, to make the slave's communication mode match the master's.

Detailed Description Text (90):
After a write cycle is initiated by the master SPROC chip, in the preferred embodiment no further read or write requests to the parallel port are possible until the current write cycle is complete. The parallel port will output a stable address and then chive the WR strobe LOW. The strobe will remain LOW for the number of WAIT states configured in the master's parallel port mode register. Valid data will be setup on the data bus, and the WR strobe will be driven HIGH after the WAIT interval, latching the data into the slave SPROC chip or peripheral. If the interface is configured in 8- or 16-bit mode, the cycle will be repeated until all bytes have been in output. After transmission of the last byte or word, the address bus and data bus will remain driven.

Detailed Description Text (105):
A SPROC chip enters boot mode when it is configured as a master SPROC chip (its MASTER input is HIGH) and the reset input (RESET) executes a LOW to HIGH transition. During boot, the parallel port is set for 8-bit mode with the maximum number of WAIT states (seven). The master SPROC chip runs an internal program, stored in its control ROM, to upload its configuration from an external 8-bit EPROM into internal RAM. The master SPROC chip will then configure any slave SPROC chips present in the system. The EPROM will be selected by a HIGH on the master SPROC chip's chip select (CS) pin, which is an output in master mode. Slave SPROC chips or memory-mapped peripherals will be selected by a LOW at this signal. In master mode, the value of the CS output is controlled by a bit set in the transmit mode register, which is the second byte of the parallel port mode register.

Detailed Description Text (107):
The parallel port incorporates a simple watchdog timer circuit to prevent any undesirable lockup states in the interface. In both master and slave modes, a read or a write flag is set (in the parallel port status register) on the initiation of a read or write operation. This flag is reset on a successful completion of the operation. If, for some reason, the host controller hangs-up in slave mode, or an invalid condition occurs in master mode, the watchdog timer will detect the situation and clear the interface flags, allowing the next operation to be accepted and executed. The watchdog timer is fixed at 256 machine cycles (1280 master clock cycles).

Detailed Description Text (108):
The watchdog timer is enabled by setting bit 16 of the parallel port mode register. SPROC reset will disable the watchdog timer. If the watchdog timer is triggered, a flag is set in the parallel port status register.

Detailed Description Text (110):
If the parallel port is performing a read or write operation in master mode, and a second write or read operation is initiated before the first I/O operation is completed, the second I/O request is locked out. A lockout flag is set in the parallel port status register.

Detailed Description Text (112):
The RTS and GPIO signals can be used for communication protocols between master and slave SPROC chips. These signals could be used as data-ready signals, requests for data, or microprocessor interrupt requests. RTS[3:0] (request to send) are four pins that function as inputs for a master SPROC chip and as outputs for a slave SPROC chip. The RTS signals of a slave SPROC can be individually set or cleared via the parallel port, as described below.

Detailed Description Text (113):
GP[3:0] are four general purpose pins that are individually configurable as either inputs or outputs. During reset, when RESET is LOW, all GPIO signals are set up as inputs. In addition to being subject to internal program control, the configuration of each GP pin, and the value of each GPIO signal configured as an output, are also individually controllable via the parallel port.

Detailed Description Text (114):
A.11 Parallel Port Registers

Detailed Description Text (115):
The parallel port utilizes five memory-mapped registers for status and control functions. The tables below list the registers and their bit definitions.

Detailed Description Text (116):
The parallel port status register, a 16-bit register, contains signal values of selected SPROC chip pins and I/O status flags. This register is updated every machine cycle (5 master clock cycles). Bits 0 through 3 contain the current signal values at the GP pins, which could individually be configured either as inputs or outputs. Similarly, bits 12 through 15 contain the current values at the RTS pins, which are inputs for a master SPROC chip and outputs for a slave. Bits 4 through 6 contain the current value of the MODE configuration.

Detailed Description Text (117):
Parallel port status register bit 10 contains the read flag, which is set while the parallel port is performing a read operation. Similarly, bit 11 contains the write flag, which is set during a write operation. (For 8- and 16-bit modes, these flags remain set until the entire 24-bit data word has been transferred.)

Detailed Description Text (118):
Bit 7 is set while the parallel port is busy servicing an I/O transaction. Bit 8 is set if the parallel port is busy in master mode and another read or write request is received. The second request will be locked out and the lockout flag set. Bit 9 is set if the watchdog timer is enabled and it detects a timeout out condition. Bits 8 and 9 can only be cleared by a SPROC reset or any write to the lockout and watchdog flag clear register.

Detailed Description Text (119):
Any write to the Watchdog/Lockout Flag Clear Register clears watchdog and/or lockout flags set in the parallel poll status register.

Detailed Description Text (120):
The parallel port input register, a 24-bit register, holds the data word received during a read operation for subsequent storage at the destination address. This register also buffers and assembles the incoming data for 8- and 16-bit modes. This register must be read by a GSP or the access port.

Detailed Description Text (121):
The parallel port GPIO/RTS Control register, a 24-bit register, is used to independently configure each GP pin as either an input or an output. It is also used to individually set and clear GP pins that are outputs, and slave SPROC chip RTS pins.

Detailed Description Text (122):
Each RTS or GPIO signal has a dedicated pair of SET and CLEAR bits in the parallel port GPIO/RTS control register. SET and CLEAR bits for RTS signals are in the low byte; SET and CLEAR bits for GPIO signals are in the mid byte. LOW values written to

both SET and CLEAR bits results in no change to the associated signal. A HIGH value at the SET bit sets the associated signal HIGH. A HIGH value at the CLEAR bit sets the associated signal LOW. If a HIGH value is written to both SET and CLEAR bits, the CLEAR dominates.

Detailed Description Text (123):
Each GPIO signal additionally has a dedicated pair of OUTPUT and INPUT bits in the high byte of the parallel port GPIO/RTS control register to configure the signal as either an output or an input. LOW values written to both OUTPUT and INPUT bits results in no change to the associated signal. A HIGH value at the OUTPUT bit configures the associated GPIO signal as an output. A HIGH value at the INPUT bit configures the associated GPIO signal as an input. If a HIGH value is written to both OUTPUT and INPUT bits, the INPUT dominates.

Detailed Description Text (124):
The master SPROC chip's parallel port mode register, a 16-bit register, controls the parallel port mode and timing.

Detailed Description Text (125):
When the master SPROC chip is reading from a slave SPROC chip or peripheral, bits 0 through 2 of the parallel port mode register (the RX MODE bits) are output at the master SPROC chip's MODE pins. Register bits 3 through 5 contain the number of WAIT states programmed for the read operation (i.e., they determine the duration of the read strobe LOW level generated by the master SPROC chip). The HIGH level between read strobes is 2 master clock cycles; this duration can be stretched to 5 master clock cycles for slower peripherals by setting bit 6 of the mode register (the RX strobe delay bit).

Detailed Description Text (126):
Similarly, when the master SPROC chip is writing to a slave SPROC chip or peripheral, bits 9 through 11 of the parallel port mode register (the TX MODE bits) are output at the master SPROC chip's MODE pins. Register bits 12 through 14 contain the number of WAIT states programmed for the write operation. The HIGH level between write strobes can be stretched for slower peripherals by setting bit 15 of the mode register (the TX strobe delay bit).

Detailed Description Text (127):
Bit 8 of the mode register is output at the master SPROC chip's CS pin. A soft reset of the parallel port, which resets the interface flags and RTS lines (but not the GPIO or MODE signals), can be initiated by setting bit 7 of this register.

Detailed Description Text (128):
While the SPROC 10 aforedescribed with a data RAM 100, a program RAM 150, a boot ROM 190, gyps 400, DFMs 600, serial ports 700, and a host port 800, is a powerful programmable signal processor in its own right, it is preferable that the SPROC be able to be programmed in a "user friendly" manner. Toward that end, a compiler system which permits a sketch and realize function is provided, as described more particularly with reference to FIG. 12. In addition, an access port 900 and a probe 1000 are provided as tools useful in the development mode of the SPROC device.

Detailed Description Text (129):
As aforementioned, the access port 900 permits the user to make changes to the program data stored in RAM 150, and/or changes to other data stored in data RAM 100 while the SPROC is operating. In other words, the access port 900 permits memory contents to be modified while the SPROC is running. In its preferred form, and as seen in FIG. 9, the access port 900 is comprised of a shift register 910, a buffer 920, a decoder 925, and a switch 930 on its input side, and a multiplexer 940 and a parallel load shift register 950 on its output side. On its input side, the access port 900 receives serial data as well as a clock and strobe signal from the development host computer. The data is arranged by the shift register 910 and stored in buffer 920 until the access port is granted time division access to the data RAM bus 125 or the program RAM bus 155. A determination as to which bus the data is to be written is made by decode block 925 which decodes the msbs of the address data stored in buffer 920. The decode block 925 in turn controls switch 930 which connects the buffer 920 to the appropriate bus. The msbs of the address data in the

buffer 920 are indicative of which RAM for which the data is destined, as the data RAM and program RAM are given distinct address spaces, as previously described.

Detailed Description Text (130):
On the output side, data received via the program RAM bus 155 or the data RAM bus 125 is forwarded via demultiplexer 940 to a shift register 950. The shift register 950 effects a parallel to serial conversion of the data so that serial data may be output together with an appropriate strobe and according to an external clock to a development host computer or the like.

Detailed Description Text (133):
As seen in FIG. 9, a plurality of SPROC devices 10a, 10b, 10c, . . . may be coupled to together as desired to provide a system of increased signal processing capabilities. Typically, the SPROC devices are coupled and communicate with each other via their serial ports 700, although it is possible for the SPROCs to communicate via their parallel host ports 800. The system of SPROCs can act as a powerful signal processing front end to a logic processor (e.g., microprocessor) 1120, or if desired, can interface directly with electromechanical or electronic components.

Detailed Description Text (139):
The preferred process of programming a SPROC is as follows. The designer must first define the signal processing application and determine design requirements. The design is then preferably placed by the designer in a signal flow diagram (using a graphic user interface). Parameters for the various blocks of the design are defined by the designer, including parameters of filters (e.g., low-pass or high pass, and cut-off frequency) and, if desired, transfer functions. Once the signal flow diagram and parameters of the blocks in the signal flow diagram are set, the diagram and parameters are automatically converted into code by the software.

Detailed Description Text (144):
After the designer captures the design and defines any necessary filters or transfer functions, the diagram and definition data files must be converted into code and a configuration file must be generated to run on the chip. The SPROCbuild utility completes this for the designer by automatically converting the diagram and data files into code, scheduling and linking the code, and generating a configuration file for the chip.

Detailed Description Text (147):
To optimize the design, the designer can modify the values of data and observe the corresponding changes in design performance. If the development system is connected to a signal generator, one can simulate various input signals and evaluate how the design reacts.

Detailed Description Text (152):
The development system comprises both hardware and software tools designed to help the designer complete the development process. The tools are designed in parallel with the SPROC chip to extract maximum efficiency and performance from the chip without compromising ease-of-use.

Detailed Description Text (154):
The SPROCboard evaluation board is a printed circuit board with one SPROC chip, digital-to-analog and analog-to-digital converters, and various communications interfaces and additional components and circuitry necessary to evaluate signal processing design performance during development. The designer can connect an oscilloscope, signal generator, or analog subsystem to the evaluation board to verify and evaluate the design. The SPROCbox interface unit provides an I/O connection between the SPROCboard evaluation board and the PC. It also connects the evaluation board to the power supply unit. The power supply unit converts AC power from a standard wall outlet to 5 VDC and 12 VDC power for use by the interface unit and evaluation board. An RS-232 cable connects the PC serial I/O port to the SPROCbox serial I/O port. A special access port cable connects the SPROCbox interface unit to the SPROCboard evaluation board. A security key connects to the PC parallel port. It enables use of the development system software. An integral power cord connects the power supply unit to the AC outlet. A positive-locking DC power

cable connects the power supply to the SPROCbox interface unit. An auxiliary DC power cable daisy chains power from the interface unit to the SPROCboard evaluation board.

Detailed Description Text (157):
The graphical design interface includes the library structure required to use the SPROCcells function library with OrCAD software. The SPROCcells function library includes cells containing DSP and analog signal processing functions for use in diagram creation. A cell is a design primitive that includes an icon required to place a function in a signal flow diagram, the code required to execute the function, and specifications for the parameters required to define the cell. The SPROCfil filter design interface supports the definition and analysis of custom digital filters. The filter design interface creates the custom code and definition data for filter cells placed in designs during diagram entry. The SPROCbuild utility converts signal flow block diagrams and their associated data files into the configuration file necessary to run on the chip. The utility interprets the output from schematic entry and incorporates associated code blocks and parameter data for cells, filter design definitions, and transfer function definitions, then schedules and links the instructions to best utilize resources on the chip. It automatically generates efficient code based on the designer's signal flow block diagram.

Detailed Description Text (159):
B.1.1 The SPROCcells Function Library

Detailed Description Text (160):
The SPROCcells function library contains over fifty predefined functions which can be used through the graphical interface of the SPROClab development system. Some cells have predefined trigger keys that aid in defining cell parameters for designs captured using OrCAD.RTM. software. Most cells include code for both inline and subroutine forms. The subroutine form of a cell performs a function identical to the corresponding inline form but includes overhead instructions that make the code in the subroutine body block re-entrant. Other subroutine versions of the cell do not include the code in their body blocks, but call the code in the body block of the first subroutine version of the cell.

Detailed Description Text (187):
Function: The dsink cell accumulates two series of input samples (each size determined by the length parameter) into two blocks of data RAM. The blocks are stored beginning at symbolic location `instance.sub.-- name.outvector1` and `instance.sub.-- name.outvector2`. Both blocks (vectors) are accessible from an external microprocessor.

Detailed Description Text (209):
Function: The dsinkrd cell accumulates two series of input samples (each size determined by the length parameter) into two blocks of data RAM. The blocks are stored beginning at symbolic location `instance.sub.-- name.outvector1` and `instance.sub.-- name.outvector2`. Both blocks (vectors) are accessible from an external microprocessor. A reset input is available: if.gtoreq.0.5, the cell is held in reset, otherwise the cell can capture a series of input samples. The done output is zero if the cell is reset or capturing input samples, else the done output is one. The done output needs to be terminated, either by another block or by a dummy module. Reset is only effective when the sink block is full.

Detailed Description Text (264):
Function: The filter cell is used for the implementation of filters designed with SPROCfil. For each instance of this cell there must be an associated filter data file produced by SPROCfil, an fdf file. This is identified with the spec parameter. An optional type parameter allows filter type verification during the compilation process.

Detailed Description Text (265):
Algorithm: Each IIR filter cell in a SPROCfll design is implemented as a cascade of biquad cells, plus a bilinear cell for odd order filters. An FIR filter cell in a SPROCfil design is split into blocks, with a default of 30 coefficients; this is a scheduler parameter.

Detailed Description Text (369):
Other cells in the function library include: ACOMPRES, AEXPAND, AGC, AMP, ANTILN, BILINEAR, BIQUAD, DECIM, DIFFAMP, DIFFCOMP, DIFF.sub.-- LDI, FIR, FWG.sub.-- NEG, FWR.sub.-- POS, GP.sub.-- IN, GP.sub.-- OUT, HARDLIM, HWR.sub.-- NEG, HWR.sub.-- POS, INTERP, INT.sub.-- LDI, INT.sub.-- RECT, INTR.sub.-- LDI, INT.sub.-- Z, MINUS, MULT, NOISE, PLL.sub.-- SQR, PULSE, QUAD.sub.-- OSC, RTS.sub.-- IN, RTS.sub.-- OUT, SCALER, SER.sub.-- IN, SER.sub.-- OUT, SINE, SINE.sub.-- OSC, STEO.sub.-- IN, STEO.sub.-- OUT, SUM2 through SUM10, TRANSFNC, UCOMPRES, UEXPAND, VCO.sub.-- SQR, and VOLTREF.

Detailed Description Text (373):
A cell is a design primitive corresponding to a specific block of SPROC description language (SDL) code. The SPROCcells function library includes many commonly used cells, and the designer can create additional cells in SDL to meet special needs. Each cell has a graphical symbol, or icon, that represents the cell and illustrates the number of inputs and outputs the cell uses. A function is inserted into the signal processing flow by placing the icon for that cell into the signal flow diagram and connecting, or wiring, the icon to other icons in the diagram.

Detailed Description Text (377):
OrCAD software requires that icons for function cells be grouped into structures called libraries. The software uses these structures to organize the cells and create menus through which the designer can access them. A library contains all of the icons for a specific grouping of functions. The functions in the SPROCcells function library are organized into a single OrCAD library. In OrCAD, parameter specifications, including cell instance names, are recorded in part fields. All cell instances have at least one part field containing the instance name. If an instance name is not specified, a default name is created.

Detailed Description Text (382):
The SPROCbuild utility is used to convert the signal flow block diagram into code and generate a chip configuration file, the utility reads the filter data file for each filter cell instance and generates the appropriate code to implement the filter as specified. The generated code uses the coefficients from the filter data file and a cascade of special filter cells to implement the filter. The special cells are provided in the SPROCcells function library, but reserved for internal use by the SPROCbuild utility.

Detailed Description Text (432):
The SPROCcells function library includes a transfer function cell so that the designer can include transfer functions in the signal processing designs. When placing this cell in a diagram, one must specify a parameter that names the file defining the transfer function.

Detailed Description Text (441):
The SPROCbuild utility provides a set of software modules that automatically converts one's design into SPROC description language (SDL) code, then uses that code to generate a configuration file for the SPROC chip and a table of symbolic references to chip memory locations. To create these files, the utility uses files produced by the SPROCview graphical design interface, the SPROCcells function library, and the SPROCfil filter design interface in the development system, and user-defined cells and transfer functions of the proper form created outside the development system.

Detailed Description Text (445):
1. The MakeSDL module integrates the output from the graphical design interface with data files from the filter design interface and user-defined transfer functions to produce a partial code package containing SDL code and data files. The module also generates instances of certain special cells to implement filter and transfer function cells. These cells are included in the SPROCcells function library but reserved for internal use.

Detailed Description Text (446):
2. The Schedule module takes the files produced by MakeSDL and adds the code blocks

for the cells used in the design (from the function library or user-defined cells) and any data files required in addition to those included in the partial code package obtained from MakeSDL. Then the Schedule module schedules the code according to on-chip resource availability and adds special "glue" cells called phantoms that provide control and synchronization functions for the general signal processors (GSPs) on the chip. These cells are included in the SPROCcells function library, but reserved for internal use. The Schedule module produces binary program and data files for the design. It also produces a file of symbolic references to chip memory locations.

Detailed Description Text (453):
Internal reserved cells included in the function library and used for special functions (i.e., to implement filters and transfer functions). This input is required.

Detailed Description Text (460):
The Schedule module takes the partial SDL code package produced by the MakeSDL module and integrates the code blocks for all necessary functions to form a complete SDL code package. It also collects all necessary data files. Then the module determines the appropriate order in which to run the code, calculates the chip resources required, and inserts the necessary phantom cells to glue the design together. Then the module converts the code package into a binary program file containing executable instructions, and an associated data file.

Detailed Description Text (461):
The Schedule module takes the following files as input: mydesign.sdl, produced by the MakeSDL module; the data files produced by the MakeSDL module; any additional data files; the SDL code blocks for function cells, function.sdl, supplied in the function library or created by the user (where function is the name of an individual signal processing function cell) and produces the following files as outputs: mydesign.spp, the binary program file and mydesign.spd, the associated data file. In addition, the Schedule module produces the symbol file (mydesign.sps) containing a table of symbolic references to SPROC chip memory locations.

Detailed Description Text (485):
A listing file can be used to verify cells. It consists of a listing of the source input and the hexadecimal Codes for corresponding data and program locations. The user can produce a listing file by invoking the SPROCbuild utility's Schedule module directly from DOS, using the invocation command line switch -l and specifying the input source file. Because the listing file is generated at compile time, outside the context of a particular instantiation of an assembly language block, it cannot include any data that is not known before the block is instantiated, i.e., any data that must come from a parameter value of a cell instance. For example, if a parameter used in the calculation of an operand value has no default value, then it cannot be known until the block is instantiated. For such operands, the operand field of the instruction is left zero, and a question mark (?) is placed immediately after. The question mark indicates that the operand value is unknown at this time. On the other hand, if a default for the parameter value has been specified, then this value is used for the instruction, and no question mark is added. Similarly, absolute addresses for instruction jumps and relocatable data references cannot be known at compile time. Whenever such an address is encountered as an operand, its absolute address with respect to the start of the block is used, and an apostrophe (') is placed immediately after. The apostrophe indicates that the address operand will be relocated.

Detailed Description Text (487):
Most cells in the SPROCcells function library include both in-line and a subroutine form code. When a cell instance occurs with the in-line form specified, the instructions for the function are instantiated as one piece of code, along with associated variable allocations. When a cell instance occurs with the subroutine form specified, the instructions for the function are instantiated as two pieces of code: one as a call block, and one as a subroutine body block. (Each piece of code may have associated variable allocations.) When subsequent instances of the same cell are specified with the subroutine form, only the call block, i.e., the piece of code necessary to call the subroutine body block, is instantiated. Only one instance

of the subroutine body block is instantiated, no matter how may times the subroutine version of the cell appears in a design. For example, if five subroutine versions of a particular cell are used in one design, the design will include five call blocks for that function, and one subroutine body block.

Detailed Description Text (495):
A time zone is a slice of time particular to a logical partition of operations on the SPROC chip. A time zone can contain any number of operations, up to the bandwidth limitations of the chip. A design may contain any number of independent time zones, up to the bandwidth limitations of the chip. Sets of operations that occur along the same logical wire (serial data path) in a design occupy the same time zone. This is analogous to the physical notion of time division multiplexing, where within a particular slice of time, anything can be accomplished so long as it does not take longer than the length of the time slice. In time division multiplexing, specific time slices are allotted to specific operations, so that a given operation can only be performed in its assigned time slice or number of time slices. Operations that require longer than the length of one time slice must be completed over multiple time slices allotted to that operation.

Detailed Description Text (500):
Turning to FIG. 10, a flow diagram of the SPROC and microprocessor development environment is seen. At 2010, using graphic entry packages such as "Draft", "Annotate", "ERC" and "Netlist" which are available from OrCad in conjunction with cell library icons such are provided from a cell library 2015 , a block diagram such as FIG. 11 is produced by the user to represent a desired system to be implemented. The OrCAD programs permit the user to draw boxes, describe instance names (e.g., multiplier 1, multiplier 2, etc. such as seen in FIG. 11 as MULT1, MULT2, . . .), describe parameters of the boxes (e.g., spec=filter 1; or upper limit=1.9, lower limit 1.9 such as seen in FIG. 11) and provide topology (line) connections. The output of the OrCad programs is a netlist (a text file which describes the instantiation, interconnect and parameterization of the blocks) which is fed to a program MakeSDL 2020 which converts or translates the netlist output from OrCad into a netlist format more suitable and appropriate for the scheduling and programming of the SPROC. Source code for MakeSDL is attached hereto as Appendix A. It will be appreciated that a program such as MakeSDL is not required, and that the netlist obtained from the OrCad programs (or any other schematic package program) can be used directly.

Detailed Description Text (502):
MakeSDL outputs SDL (SPROC description language) netlist files and data files. The data files represent data values which are intended for the SPROC data RAM and which essentially provide initial values for, e.g., filter coefficients and source blocks. For functions not located in the cell library, a text editor 2035 can be used to generate appropriate SDL and data files. Those skilled in the art will appreciate that any text editor can be used. What is required is that the output of the text editor be compatible with the format of what the scheduler /compiler 2040 expects to see.

Detailed Description Text (503):
Both the netlist and data files output by the MakeSDL program are input to a scheduling /compiling program as indicated at 2040. In addition, a cell library 2015 containing other SDL files are provided to enable the scheduler /compiler to generate desired code. Among the signal processing functions provided in the cell library are a multiplier, a summing junction, an amplifier, an integrator, a phase locked loop, an IIR filter, a FIR filter, an FFT, rectifiers, comparators, limiters, oscillators, waveform generators, etc. Details of the scheduler /compiler are described in more detail hereinafter, and source code for the scheduler /compiler is attached hereto as Appendix M.

Detailed Description Text (504):
The output of the scheduler /compiler contains at least three files: the .spd (SPROC data) file; the .spp (SPROC program) file; and the .sps (SPROC symbol) file. The SPROC data file contains initialization values for the data locations of the SPROC (e.g., 0400 through ffff), which data locations can relate to specific aspects of the SPROC as discussed above with reference to the SPROC hardware. The SPROC program

file contains the program code for the SPROC which is held in SPROC program RAM (addresses 0000 to 03 ff) and which is described in detail above with reference to the SPROC hardware. The SPROC symbol file is a correspondence map between SPROC addresses and variable names, and is used as hereinafter described by the microprocessor for establishing the ability of the microprocessor to control and/or communicate with the SPROC. If desired, the scheduler /compiler can produce other files as shown in FIG. 10. One example is a .spm file which lists the full file names of all included files.

Detailed Description Text (505):
As aforementioned, the scheduler /compiler produces a symbol file (.sps) for use by the microprocessor. Depending upon the type of microprocessor which will act as a host for the SPROC, the symbol file will be translated into appropriate file formats. Thus, as shown in FIG. 10, symbol translation is accomplished at 2050. Source code in accord with the preferred embodiment of the invention is provided in Appendix C for a symbol translator which translates the .sps file generated by the scheduler /compiler 2040 to files which can be compiled for use by a Motorola 68000 microprocessor. In accord with the preferred embodiment, the symbol translator 2050 generates to files: a .c (code) file, and a .h (header) file. The code file contains functions which can be called by a C program language application. The header file contains prototypes and symbol definitions for the microprocessor compiler hereinafter described.

Detailed Description Text (506):
Returning to the outputs of the scheduler /compiler 2040, the data and program files are preferably fed to a program MakeLoad 2060 (the source code of which is provided as Appendix D hereto. The MakeLoad program merges the spp and spd into a file (.blk) which is in a format for the microprocessor compiler and which can be used to initialize (boot) the SPROC. Of course, if desired, the .blk file can be loaded directly into a microprocessor if the microprocessor is provided with specific memory for that purpose and a program which will access that memory for the purpose of booting the SPROC.

Detailed Description Text (507):
The Makeload program also preferably outputs another file .lod (load) which contains the same information as the .blk (block) code, but which is used by the SPROCdrive interface 2070 to boot the SPROC in stand-alone and development applications. Details regarding the SPROCdrive interface are discussed below. Another input into the SPROCdrive program is the symbol file (.sps) generated by the scheduler /compiler 2040. This allows the SPROCdrive program to configure the SPROC and control the SPROC symbolically. In particular, if it was desired to read the output of a particular block, a command "read blockname.out" can be used. The .sps file then provides the SPROC address corresponding to the symbol blockname.out, and the SPROCdrive interface then sends a read and return value command to the SPROC 10 via the SPROCbox 2080. The function of the SPROCbox is to provide an RS232 to SPROC access port protocol conversion, as would be evident to one skilled in the art.

Detailed Description Text (511):
SDL is a block-oriented language that supports hierarchical designs. Blocks may be either primitive or heirarchical.

Detailed Description Text (512):
Primitive blocks also called asmblocks contain hardware-specific coding analogous to the firmware in a microprocessor system. Primitive blocks are written in assembly language. They may not contain references to other blocks.

Detailed Description Text (513):
Code for signal processing functions is written at the primitive level. These primitive blocks comprise the SPROCcells function library. They are optimized for the hardware and efficiently implemented to extract maximum performance from the SPROC chip. Other primitive blocks include the glue blocks or phantoms required to provide control ;and synchronization functions for the multiple general signal processors (GSPs) on the SPROC chip.

Detailed Description Text (514):

Hierarchical blocks contain references to other blocks, either primitive or hierarchical. The sequence (i.e.,firing order) and partitioning (i.e., allocation over the GSPs and insertion of phantom blocks) of the referenced blocks in a hierarchical block is automatically determined.

Detailed Description Text (516):
Two types of special-purpose hierarchical blocks are also available: sequence blocks and manual blocks.

Detailed Description Text (517):
A sequence block is a hierarchical block that is not automatically sequenced. The order of the references contained in a sequence block specifies the firing order of the referenced blocks.

Detailed Description Text (518):
A manual block is a hierarchical block that is neither automatically sequenced nor partitioned. As with the sequence block, the order of block references in a manual block specifies the firing order of referenced blocks. In addition, referenced blocks are not partitioned, and no phantom blocks are inserted.

Detailed Description Text (519):
A block contains a block name, block definition, and block body. The block name identifies the block for reference by hierarchical blocks. The block definition contains an optional list of parameters; a port list declaring the block's input and output signals; optional general declarations for wires, variables, symbols, aliases, time zones, compute lines, and ports; optional verify statements; and optional duration statements (primitive blocks only)

Detailed Description Text (520):
The block body contains references to other blocks (hierarchical blocks only) or assembly lines (primitive blocks and manual blocks only).

Detailed Description Text (521):
C.2 Compiling SDL Files

Detailed Description Text (522):
SDL files are compiled by the SPROCbuild utility in the SPROClab development system. The utility includes three modules: the MakeSDL module, the Schedule module, and the MakeLoad module.

Detailed Description Text (523):
The MakeSDL module prepares a top-level SDL file that completely describes the signal processing design using the netlist of the signal flow block diagram, primitive blocks from the function library, and other code and data files

Detailed Description Text (524):
The Schedule module takes the top-level SDL file and breaks the file apart based on the resource and synchronization requirements of the blocks within the file. Resource requirements include program memory usage, data memory usage, and GSP cycles. Synchronization requirements include the determination of how and when blocks communicate data, and whether a block is asynchronous and independent of other blocks in the design.

Detailed Description Text (525):
After breaking up the file to accommodate resource and synchronization requirements, the Schedule module partitions the file by blocks and locates the blocks to execute on the multiple GSPs on the SPROC chip using a proprietary partitioning algorithm. The module inserts phantom blocks as necessary to control the synchronization of the GSPs as they execute the design.

Detailed Description Text (527):
The MakeLoad module converts the partitioned SDL file into a binary configuration file to run on the chip.

Detailed Description Text (532):

Expressions : An expression is a statement of a numerical value. An expression can be simply a number or identifier (like a register name), or a combination of numbers, symbols, and operators that evaluates to a numerical value. Expressions in a block are evaluated when the block is instantiated. Expressions may be used wherever a number is required. The operand field of most instructions may contain an expression. Initial values for variables may be declared as expressions . The table below lists the valid operators that may be used in expressions .

Detailed Description Text (533):
Expressions may include identifiers, like parameter names, symbols, and variable, wire, or port names. When translating identifiers to evaluate an expression, if the identifier cannot be found in the current block instance, block instances in successively higher levels are searched until the identifier is found. The identifier is evaluated in the context of the block instance in which it is found.

Detailed Description Text (534):
Numbers used in expressions may be real, integer, hex, or binary numbers. An expression containing a real number evaluates to a real number and may be assigned to the FIXED data type. An expression containing no real numbers (only integer, hex, or binary numbers) evaluates to an integer number and may be assigned to an INTEGER data type.

Detailed Description Text (541):
Parameters: A parameter is an identifier or string that provides a means of customizing an occurrence, or instance, of a block. Parameter values may be passed by a hierarchical block to a lower level hierarchical block. They can also provide immediate values and array sizes for primitive blocks . Parameters are declared in the optional parameter listing for a block. A default value for a parameter may be declared. When a block is instantiated, any parameter values supplied for the instance override the default values. Parameters reserve no storage. Parameter names should begin with a percent sign (%).

Detailed Description Text (544):
Symbols: A symbol is a series of characters (an identifier or a string) that names an expression. The symbol for an expression may be used in other expressions . Appropriately chosen symbol names may be used to make SDL code more readable. Symbols are declared using symbol statements. Symbols may be used within blocks as an initial value for a variable or wire, for example. A symbol may also be passed via a parameter value assignment to an instance within a hierarchical block.

Detailed Description Text (545):
Time Zones: A time zone declaration is required for every signal source block, like the primitive block for a signal generator function, or a serial input port function. The time zone statement declares a time zone name for the block, and optionally, a sample rate for that zone, in samples per second. A sample rate need only be given in one such time zone statement of multiple blocks that declare the same time zone name. The time zone name is used to determine synchronization requirements of blocks . Time zones which have different names are taken to be asynchronous, even if they have the same sample rate.

Detailed Description Text (546):
Variables: Variables are declared using variable statements. Variables may be declared for hierarchical or primitive blocks . A variable may be declared, and an initial value for the variable may also be declared. The value may be a number (or expression) or a string (or an expression with no value assigned to it) that identifies a file containing the initial values for the variable. If the string (file name) entered as the value for a variable includes a period (.), it must be enclosed in single or double quotes. Numbers in the file must be delimited by spaces, tabs, or new lines, and may be real, hexadecimal, integer, or binary numbers. If the file contains fewer values than are required by the variable, any missing values are zero filled. Variables may be scalar or dimensioned as single-dimension arrays. If an initial value is declared as a number (or expression) that value is duplicated for array variables.

Detailed Description Text (547):

Wires: A wire is a signal between primitive blocks in a design. Wires are declared using the wire statement. Wires may be declared in hierarchical blocks only. A wire may be declared, and an initial value for the wire may also be declared. The value may be a number (or expression) or a string (or an expression with no value assigned to it) that identifies a file containing the initial value for the wire. The wire value is by default type FIXED. Wire values are scalar only.

Detailed Description Text (553):
5. Write the output of a block to memory each time the block is executed. The software-directed probe used for debug is triggered by writing block output to memory, and will not function properly if blocks are executed without their outputs being written to memory.

Detailed Description Text (570):
A symbol statement defines a symbol name for use in expressions. The optional micro keyword makes the symbol available for access from a microprocessor for applications using SMI.

Detailed Description Text (601):
D. The SPROC Compiler

Detailed Description Text (602):
Returning to details of the scheduler /compiler 2040, the basic function of the scheduler /compiler 2040 is to take the user's design which has been translated into a scheduler /compiler understandable format (e.g., SPROC Description Language), and to provide therefrom executable SPROC code (.spp), initial data values (.spd), and the symbol file (.sps). The preferred code for the scheduler /compiler is attached hereto as Appendix M, and will be instructive to those skilled in the art.

Detailed Description Text (603):
The scheduler /compiler does automatic timing analysis of each design provided by the user, allocating the necessary SPROC resources to guarantee real-time execution at the required sample rate. In order to guarantee real-time execution, the scheduler /compiler preferably performs "temporal partitioning" (although other partitioning schemes can be used) which schedules processors in a round-robin fashion so as to evenly distribute the compute power of the multi-GSP SPROC. Each GSP picks up the next sample in turn, and executes the entirety of the code in a single time zone (i.e., that part of the user's design which runs at the same sample clock). Additional information regarding time zones can be obtained by reference to U.S. Pat. No. 4,796,179 to Lehman et al. which provides time zones for a microprocessor based system.

Detailed Description Text (604):
The scheduler /compiler 2040 also insert "phantom blocks" into the user's design which supply the necessary system "glue" to synchronize processors and input/output, and turn the user's design specification into executable code to effect a custom operating system for the design. Preferred code for the phantom blocks is found attached hereto as Appendix E (Scheduler /Compiler Phantom Block Source Code).

Detailed Description Text (605):
Because it is possible for a block which the user has designated to have a varying execution time, the GSPs running common code under temporal partitioning could conceivably collide or get out of sequence. Phantom blocks called "turnstiles" are inserted at every sample period's worth of code to keep the GSPs properly staggered in time. By computing and using maximum and minimum durations rather than a maximum duration and an assumed minimum duration of zero, the turnstiles may be placed to optimize the code variability. The scheduler /compiler code provided in Appendix M, however, does not optimize in this manner. Also, output FIFOs are created whose size depends on code execution time variability. These output FIFOs can also be optimized.

Detailed Description Text (606):
In temporal partitioning, a GSP can overwrite a signal memory location with its new value before the old value has been used by another GSP which requires that value. In order to prevent this overwriting problem, phantom blocks which create "phantom

copies" of signal values are inserted. A different manner of solving this problem is to cause each GSP to maintain its own private copies of signal values, with phantom blocks automatically added, which for each signal, writes successive values to an additional single memory location so that it may be probed at a single memory address.

Detailed Description Text (607):
The scheduler /compiler supports asynchronous timing as well as decimation and interpolation. Decimation and interpolation are accomplished within temporal partitioning by "blocking" the signal values into arrays, and operating on these arrays of values rather than on single signal values. Thus, for example, in decimating by four, four input samples are buffered up by the input data flow manager. The code blocks before the decimator are looped through four times, along with any filtering associated with the decimation, and then the code after the decimator is run once.

Detailed Description Text (608):
Various design integrity checks are performed by the scheduler, such as determining if multiple inputs to a cell have incompatible sample rates, or if any inputs have been left "floating". The scheduler /compiler supports feedback loops within a design, and automatically detects them.

Detailed Description Text (609):
A powerful parameter-passing mechanism in the scheduler /compiler allows each instance of a cell to be customized with different parameters. Parameter values need not be absolute, but can be arbitrary expressions utilizing parameters of higher level cells. The scheduler /compiler provides for cells to verify that their parameter values are legal, and to issue compile-time error messages if not.

Detailed Description Text (610):
Arbitrary design hierarchy is supported by the scheduler /compiler, including multiple sample rates within hierarchical blocks. Using only a schematic editor (e.g., the OrCad system), users may build their own hierarchical composite cells made up of any combination of library primitive cells and their own composite cells. Details of component cells may be hidden, while providing any necessary SPROCdrive interface/SPROClink microprocessor interface access to selected internal memory locations. Composite cells may also be built which provide access to internal memory locations directly through the composite cell's input/output, allowing efficient, directly wire control of parameters.

Detailed Description Text (611):
A high level flow diagram of the compiler preferably used in conjunction with the SPROC 10 of the invention is seen in FIG. 12. When the user of the development system wishes to compile a design, the user runs the compiler with an input file containing the design. The compiler first determines at 1210 which of its various library blocks (cell library 2015) are needed. Because some of the library blocks will need sub-blocks, the compiler determines at 1212 which sub-blocks (also called child blocks) are required and whether all the necessary library block files can be read in. If they can, at 1220 the compiler creates individual instances of each block required, since the same block may be used more than once in a design. Such a block may be called with different parameters which would thereby create a different version of that block. The instances generated at step 1220 are represented within the compiler data structures as a tree, with the top level block of the user's design at the root of the tree. At 1230, the compiler evaluates the contents of each instance, and establishes logical connects between the inputs and outputs of child instances and storage locations in higher level instances. In evaluating an instance, the compiler determines code and data storage requirements of that instance, and assembles the assembly language instructions which comprise the lowest level child instances. At 1240, the compiler sequences the instances by reordering the list of child instances contained in each parent instance. This is the order in which the set of program instructions associated with each lowest level child instance will be placed in the program memory 150 of the SPROC 10. To do this, the compiler traces forward from the inputs of the top level instance at the root of the tree, descending through child blocks as they are encountered. When all inputs of an instance have been reached, the instance is set as the next child instance in the

sequence of its parent instance. Feedback loops are detected and noted. At 1250, the compiler partitions the design over multiple GSPs. Successive child instances are assigned to a GSP until adding one more instance would require the GSP to take more than its allowed processing time; i.e. one sample period. Succeeding child instances are assigned to a new GSP, and the process continues until all the instances are assigned to respective GSPs. As part of the partitioning step 1250, the compiler inserts instances of phantom blocks at the correct points in a child sequence. Phantom blocks are blocks which are not designated by the user, but which are necessary for the correct functioning of the system; e.g. blocks to implement software FIFOs which pass signals form one GSP to the next GSP in the signal flow. At step 1260, the compiler re-evaluates the instances so that the phantom block instances added at step 1250 will be fully integrated into the instance tree data structure of the compiler. Then, at 1270, the compiler generates program code (.spp) by traversing the instance tree in the sequence determined at step 1240, and when each lowest level child instance is reached, by outputting to a file the sequence of SPROC instructions assembled for that instance. It also outputs to a second file desired initialization values (.spd) for the data storage required at each instance. It further outputs to a did file the program and data locations referenced by various symbolic names (.sps) which were either given by the user or generated automatically by the compiler to refer to particular aspects of the design. As aforementioned, additional details of the scheduler /compiler may be seen in the preferred code for the scheduler /compiler which is attached hereto as Appendix M.

Detailed Description Text (613):
Referring now to the microprocessor side of FIG. 10, a C compiler and linker available from Intermetrics (including a debugger "XDB" and a compiler/linker "C Tools") is shown for a microprocessor (logic processor). The inputs to the C compiler include the symbol translation files (.c and .h) provided by the symbol translator 2050, the SPROC boot file (.blk) provided by the MakeLoad software 2060, functions provided by the SPROC C library 2110 hereto, and either manually generated text editor inputs from text editor 2035 or automatically generated code such as might be generated according to the teachings of U.S. Pat. No. 4,796,179 from a block diagram. Because of the code provided by the symbol translator 2050, source code from the text editor can refer symbolically to variables (e.g., filt1.out) which have been compiled by the SPROClab system. This is critical for the ability of the microprocessor to interface with the SPROC; i.e., for the microprocessor to obtain information via the host or other port from various locations in the SPROC RAM.

Detailed Description Text (614):
The SPROC C function library routines are provided to convert SPROC data types to C compatible data types. The SPROC boot file provided to the C compiler and linker 2100 by the MakeLoad routine 2060 is not particularly processed by the compiler, but is fed into a memory block of the microprocessor's memory space.

Detailed Description Text (615):
The output of the compiler/linker 2100 can be used directly to program the microprocessor 2120, or as shown in FIG. 10 can be provided to a microprocessor emulator 2110. Microprocessor emulator 2110 available from Microtek helps in the debugging process of the microprocessor. As the emulator is not a required part of the system, additional details of the same are not provided herewith. As shown in FIG. 10, the programmed microprocessor 2120 interfaces with the SPROC 10 through the host (parallel) port of the SPROC, although information can be obtained in a serial fashion from a SPROC access port if desired.

Detailed Description Text (616):
As aforementioned, the compiler/linker 2100 for the microprocessor receives code from a text editor or an automatic code generating system. To read and write sample data values, icons are placed on the signal processor block diagram (an example of which is shown in FIG. 11), and the symbol names which might be read from or written to by the microprocessor are made known to the microprocessor compiler/linker by the symbol translator. If the user wishes to read or write signal processor block diagram parameter values (e.g., gain of ampl=x), the user references the symbol name in the microprocessor source code (i.e., the user uses the text editor).

Detailed Description Text (617):
In accord with another aspect of the invention, code for the microprocessor may be automatically generated rather than being generated by the user via the text editor. In automatically generating code for the microprocessor, a block diagram of the microprocessor functions can be entered in a manner similar to that described above with reference to the signal processor block diagram. Then, utilizing the teachings of U.S. Pat. No. 4,796,179, code may be generated automatically. Where automatic programming via block diagram entry of both the signal processor and microprocessor is utilized, reading and writing by the microprocessor of sample data values of the SPROC is accomplished as before (i.e., icons are placed on the signal processor block diagram and the symbol names which might be read from or written to by the microprocessor are made known to the microprocessor compiler/linker by the symbol translator.) However, the reading or writing by the microprocessor of signal processor parameter values is preferably accomplished by providing "virtual" wires between the microprocessor and SPROC blocks. Because the virtual wires are not real wires, no storage is allocated to the virtual wires by either the SPROC or the microprocessor during compilation. However, the location (e.g., ampl gain) to which the virtual wire refers is placed in the .sps file such that the symbol translator 2050 makes it known to the automatic microprocessor compiler. In this manner, the symbolic reference to the parameter is the same for both the SPROC and microprocessor compilers and this permits the microprocessor to read or write that parameter.

Detailed Description Text (618):
Where graphic entry and automatic programming are used for both the microprocessor and SPROC, some means for distinguishing what is to be processed by the microprocessor and what is to be processed by the SPROC is required. A simple manner of distinguishing between the two is to require user entry which will define a block as a block to be executed by the SPROC (e.g., block.spr) or a block to be executed by the microprocessor (e.g., block.mic). Where it is desired to provide blocks which will be executed by both the SPROC and the microprocessor (possibly also including virtual wires), a hierarchical block should be provided. The hierarchical block will contain child blocks which will be designated as .spr or .mic blocks as discussed above.

Detailed Description Text (619):
Another manner of distinguishing what is to be processed by the microprocessor and what is to be processed by the SPROC is to segment the tasks by the sample rate at which the block is functioning, with the relatively slow sampling rate tasks being handled by the microprocessor and the relatively fast sampling rate tasks being handled by the signal processor. Of course, if all blocks are predefined (i.e., are contained in a library), the precoded library code divides the code into code intended for the SPROC and code intended for the microprocessor. Regardless, where graphic entry for both signal processing and logic processing is permitted, the graphic entry eventually results in separate automatic compilation for both the SPROC and the microprocessor, with the SPROClab compiler again providing the necessary symbol table for incorporation during compilation of the microprocessor code.

Detailed Description Text (621):
The SPROClink microprocessor interface (SMI) is a set of components used to develop microprocessor applications in ANSI C that include the SPROC chip as a memory mapped device. With the components of SMI, one can create microprocessor applications that separate the logic processing tasks that run best on a microprocessor from the real-time signal processing tasks that run best on the SPROC chip. Partitioning the design in this way increases the performance and efficiency of the application.

Detailed Description Text (622):
The SPROC chip communicates with the microprocessor at high speed via the SPROC chip parallel port and appears as a memory mapped device occupying 16K bytes of microprocessor memory space. SPROC chip memory is 4K of 24-bit words that map to the microprocessor as 32-bit words.

Detailed Description Text (625):
SMI includes a symbol translator (SymTran) and the SPROC C function library

(sproclib.c).

Detailed Description Text (627):
The SPROC C function library contains the source code and header files for basic functions required to access the SPROC chip from a microprocessor. The library includes the SPROC chip load, reset, and start functions, as well as the data conversion functions required for the microprocessor to correctly access and interpret the 24-bit fixed-point data type native to the SPROC chip.

Detailed Description Text (632):
In the embedded system development environment, one must translate the symbol file produced by the SPROCbuild utility into the data structure needed to provide microprocessor access to SPROC chip memory addresses; copy the configuration file, the data structure, and all relevant file sections from the SPROC C function library into the applications work area; and create the microprocessor application. In addition, one must also map the SPROC chip(s) into the microprocessor's memory.

Detailed Description Text (633):
It should be noted that aspects of the microprocessor application depend on output from the signal processing design development process. If one develops the portion of the microprocessor application that deals with the SPROC chip in parallel with the signal processing design, it is important to understand the relationship between the two processes and the dependencies described herein. Otherwise, changes made to the signal processing design may require changes to the microprocessor application.

Detailed Description Text (636):
The configuration file is produced by MakeLoad, a module of the SPROCbuild utility in the SPROClab development system. It includes the program and data that comprise the signal processing design. As a default, the SPROCbuild utility produces a standard configuration file in Motorola S-record format that can be loaded to the SPROC chip from the SPROCdrive interface. This standard configuration file is called a load file and has the file extension .lod. Because the configuration file used by SMI will be compiled by a C compiler and loaded from a microprocessor, the standard S-record format configuration file cannot be used. A special configuration file, called a block file, is required. The block file contains the C code describing the SPROC chip load as an initialized array of data, and it has the file extension .blk. The SPROCbuild utility will produce the configuration file as a block file in addition to a load file if the invocation line switch for the MakeLoad module is entered when running the SPROCbuild utility.

Detailed Description Text (637):
The symbol file is produced by the Schedule module of the SPROCbuild utility. The standard symbol file has the file extension sps. It provides access to memory addresses on the SPROC chip using symbolic names instead of direct memory references. Because the symbol file used by SMI must be included in C programs and compiled by a C compiler, the file must be in a different format than the standard symbol file. To create this special version of the symbol file, the symbol translator (SymTran) takes as input the symbol file generated by the SPROCbuild utility and produces C header and code files that create a data structure mirroring the symbol file and including all necessary variable declarations. The symbol translator produces one header file and one code file for each signal processing design.

Detailed Description Text (639):
In order for a signal processing design created in the SPROClab development system to be usable in a microprocessor application, the microprocessor must have access to the values of variables in the design running on the SPROC chip. Special cells in the SPROCcells function library provide this access by flagging specific nodes in the design for microprocessor visibility. This "micro" keyword triggers the symbol translator to make external C references to the associated symbols available. Only symbols with this micro keyword are available to the microprocessor.

Detailed Description Text (644):
In a microprocessor application, the block file is available as a block of data that can be downloaded to the memory mapped SPROC chip using a C function call and the

block variable. The sproc.sub.-- load function included in the SPROC C function library will download the block file to the SPROC chip.

Detailed Description Text (653):
E.9 Using the SPROC C Function Library

Detailed Description Text (654):
The SPROC C function library (sproclib.c) includes the basic functions necessary to allow the microprocessor to control the SPROC chip. The SPROC C function library includes source modules that determine the byte ordering supported by SMI, define SPROC data types, provide functions to convert C's data types to and from SPROC data types, and provide functions for the microprocessor to load the SPROC chip and start execution of the signal processing design. All modules are supplied as source and must be compiled and linked in the embedded system development environment. Not all modules are, required for every application. The user may select the specific modules needed for a particular application and compile and link only those. If the embedded system design environment supports libraries, one may compile all modules and build them into a library from which one can reference selected modules for linking in a specific application.

Detailed Description Text (660):
SMI supports applications using both Motorola-type (little endian) and Intel-type (big endian) byte ordering. The default byte ordering is Motorola-type. To change the byte ordering, one must edit the file sprocdef.h, or use the #define INTEL statement or the define switch on the C compiler. To edit the file sprocdef.h, comment out the lines relating to Motorola-type byte ordering, and uncomment the lines that provide support for Intel-type byte ordering.

Detailed Description Text (671):
Turning to FIG. 11, a block diagram is seen of a low frequency impedance analyzer. The analyzer includes several multipliers 2201, 2203, 2205, 2207, 2209, 2211, two scalers, 2214, 2216, two integrators 2220, 2222, two hard limiters 2224, 2226, two full wave rectifiers 2230, 2232, two filters 2234, 2236, two amplifiers 2238, 2240, two summers 2242, 2244, three arrays (sink blocks) 2246, 2248, 2250, an oscillator 2252, a serial input 2253, two serial outputs 2254, 2255 and two microprocessor software interface output cells 2256, 2258, and one microprocessor interface input cell 2260. Each block has a library name (e.g., SCALER, MULT, SUM2, FILTER, etc.), an instance name (e.g., SCALER1, MULT2, etc.), and at least one terminal, and many of the blocks include parameters (e.g., shift=, upper=, spec=, freq=, etc.). The wires between terminals of different blocks carry data sample values (as no virtual wires are shown in FIG. 11). The serial input 2253 receives data from external the SPROC at a high frequency sample data rate, and the data is processed in real time in accord with the block diagram. The SPROC outputs two values external to the SPROC and microprocessor (i.e., out the serial ports) as a result of the SPROC processing. Information to be provided to or received from the microprocessor is sent or received via the microprocessor software interface cells 2256, 2258, and 2260. In particular, when the microprocessor writes to the location of cell 2260, cell 2260 causes the arrays 2246, 2248 to collect data and to provide a signal to microprocessor software interface output cells 2256 and 2258 when filled.

Detailed Description Text (672):
With the block diagram so provided on an OrCad graphic interface, and in accord with the above description, after translation by the MakeSDL file, the scheduler /compiler provides a program file (yhpdual.spp) and a data file (yhpdual.spd) for the SPROC, and a symbol file (yhpdual.sps) for the symbol translator and microprocessor and for the SPROCdrive interface. The program, data, and symbol files are attached hereto as Appendices F, G, and H. In addition, the yhpdual.spp and yhpdual.spd files are processed by the MakeLoad program which generates the yhpdual.blk file which is attached hereto as Appendix I.

Detailed Description Text (673):
In order to completely implement the low frequency impedance analyzer such that it may be accessed by the microprocessor, the microprocessor is provided with C code. An example of C code (Maintest.C) for this purpose is attached hereto as Appendix J. Of course, similar code could be generated in an automatic fashion if an automatic

microprocessor code generator were to be utilized. As provided, the C code attached as Appendix J calls yhpdual.c and yhpdual.h which are the translated files generated by the symbol translator from the yhpdual.spp file generated by the SPROC scheduler /compiler. Attached hereto as Appendices K and L are the yhpdual.h and yhpdual.c files. Thus, the Maintest.C as well as the yhpdual.h and hypdual.c files are provided in a format which can be compiled by the microprocessor compiler.

Detailed Description Text (674):
There have been described and illustrated herein architectures and methods for dividing processing tasks into tasks for a programmable real time signal processor and tasks for a decision-making microprocessor interfacing with the real time signal processor. While particular embodiments of the invention have been described, it is not intended that the invention be limited thereto, as it is intended that the invention be as broad in scope as the art will allow and that the specification be read likewise. Thus, while particular hardware and software have been described, it will be appreciated that the hardware and software are by way of example and not by way of limitation. In particular, while a 68000 microprocessor and C compiler for the 68000 microprocessor have been described, other processors (i.e., not only "microprocessors"), and/or other types of code (e.g., FORTRAN, PASCAL, etc.) could be utilized. AH that is required is that the symbol table code (.sps) generated by the SPROClab development system be in a format for compilation by the processor compiler, and that, where provided, the boot file (.blk) be in a format for compilation by the processor compiler or in a format for storage by the processor. Similarly, while a particular real time signal processor (the SPROC) has been described, it will be appreciated that other similar type signal processors can be utilized provided that the signal processor is directed to signal processing rather than logic processing; i.e., the signal processor should have a non-interrupt structure where dam flow is through central memory. Further, while a system which provides the realization of a high level circuit in a silicon chip from simply a sketch on a graphic user interface has been described for at least the real time signal processor, it will be appreciated that the text editor could be used to replace the graphic entry, and that while the system would not be as convenient, the graphic entry is not absolutely required. Similarly, the text editor could be eliminated and the system could work only from the graphic entry interface. Other readily evident changes include: an expanded or a different cell library; different graphic user interfaces; the provision of a scheduler /compiler for the SPROC which is directly compatible with the graphic user interface (rather than using a translator such as MakeSDL); and the provision of different software packages. It will therefore be appreciated by those skilled in the art that yet other modifications could be made to the provided invention without deviating from its spirit and scope as so claimed. ##SPC1## ##SPC2## ##SPC3## ##SPC4## ##SPC5## ##SPC6## ##SPC7## ##SPC8## ##SPC9## ##SPC10## ##SPC11## ##SPC12## ##SPC13##

Detailed Description Paragraph Table (7):

Parellel Port Registers REGISTER ADDRESS	REGISTER NAME	READ/WRITE
4FB	Lockout and watchdog flag write clear	
4FC	Parallel port status register	read
4FD	Parallel port input register	read
4FE	Parallel port GPIO/RTS write control register	
4FF	Parallel port mode register	write

Parallel Port Register Bit Definitions BIT	REGISTER 4FC	REGISTER 4FE	REGISTER 4FF
0	GP[0] INPUT SETRTS[0]; RX MODE[0]		
1	GP[1] INPUT SET RTS[1] RX MODE[1]		
2	GP[2] INPUT SET RTS[2] RX MODE[2]		
3	GP[3] INPUT SET RTS[3] RX WAIT STATES [0]		
4		MODE[0] CLEAR RTS[0] RX WAIT STATES [1]	
5		MODE[1] CLEAR RTS[1] RX WAIT STATES [2]	
6		MODE[2] CLEAR RTS[2] RX STROBE DELAY	
7		PARALLEL PORT BUSY CLEAR RTS[3]	PARALLEL PORT SOFT FLAG RESET
8	LOCK OUT FLAG SET GPIO[0] ##STR1##		
9	WATCHDOG FLAG SET GPIO[1] TX MODE[0]		
10	READ FLAG SET GPIO[2] TX MODE[1]		
11	WRITE FLAG SET GPIO[3] TX MODE[2]		
12	RTS[0] INPUT CLEAR GPIO[0] TX WAIT STATES [0]		
13	RTS[1] INPUT CLEAR GPIO[1] TX WAIT STATES [1]		
14	RTS[2] INPUT CLEAR GPIO[2] TX WAIT STATES [2]		
15	RTS[3] INPUT CLEAR GPIO[3] TX STROBE DELAY		
16	N/A OUTPUT GPIO[0] WATCHDOG ENABLE		
17	N/A OUTPUT GPIO[1] N/A		
18	N/A OUTPUT GPIO[2] N/A		
19	N/A OUTPUT GPIO[3] N/A		
20	NA/ INPUT GPIO[0] N/A		
21	N/A INPUT GPIO[1] N/A		
22	N/A INPUT GPIO[2] N/A		
23	N/A INPUT GPIO[3] N/A		

Detailed Description Paragraph Table (8):

Port Signal Definitions SIGNAL TYPE* DESCRIPTION ADDRESS[15:0] O(M)I(S) ADDRESS BUS Parallel ##STR2## I BUS GRANT causes the SPROC chip to three-state the address and data buses, and MODE pins, when LOW. ##STR3## O PARALLEL PORT BUSY is set LOW when an I/O operation is occurring, set HIGH when completed. Also reset HIGH by watchdog timer if a timeout occurs. ##STR4## Tied LOW. ##STR5## O(M)I(S) CHIP SELECT signal. A slave SPROC chip is selected ##STR6## generates this signal as an output, expecting to select a ##STR7## ROM (containing every slave SPROC chip's configuration) by setting it HIGH. DATA[23:0] I/O PARALLEL PORT DATA BUS--24-bit input/output/three-statable bidirectional bus. ##STR8## O DATA TRANSFER ACKNOWLEDGE. In slave mode, ##STR9## LOW and the SPROC chip has completed the data ##STR10## This output is always HIGH for a master SPROC chip. EADDRESS[1:0] O(M)I(S) EXTENDED ADDRESS specifies which portion of the full 24-bit word is currently being transferred in 8- and 16-bit modes. GP[3:0] I/O GENERAL PURPOSE I/O lines, individually configurable as either input or output. Can be used to interface SPROC chips with each other or with an external controller as data-ready, microprocessor interrupt requests, etc. Controlled and configured by a write to parallel port GPIO/RTS control register. MASTER I MASTER causes SPROC chip to operate in master mode when HIGH, and in slave mode when LOW. MODE[2:0] O(M)I(S) MODE[0] differentiates between full 24-bit mode (HIGH) and partial (8- or 1 6-bit) modes (LOW). MODE[1] differentiates between 8-bit mode (HIGH) and 16-bit mode (LOW) for partial data transfers. MODE[2] specifies whether the first 8- or 16-bit transmission contains the lsb (HIGH) or the msb (LOW). ##STR11## Tied LOW. ##STR12## O(M)I(S) READ strobe generated by master SPROC chip or ##STR13## ##STR14## to successfully complete the READ; programmed WAIT ##STR15## ##STR16## returns HIGH. ##STR17## I ##STR18## clock cycles. after power and clock have stabilized. This input is a Schmitt trigger type which is suitable for use with an RC time constant to provide power-on reset. ##STR19## will force address, extended address, and SPROC select ##STR20## HIGH. Slave SPROC chips connected to the bus will then be deselected and have driven inputs. MODE[2:0] will be configured for 8-bit boot mode with msb byte first and zero WAIT states. The data bus will be driven. RTS[3:0] I(M)O(S) REQUEST TO SEND flags. These pins are outputs for slave SPROC chips and inputs for master SPROC chips. Can be used to interface slave with master or external controller as data-ready, microprocessor interrupt requests, etc. Controlled and configured by write to parallel port GPIO/RTS control register. ##STR21## O(M)I(S) WRITE strobe generated by master SPROC chip or ##STR22## ##STR23## to successfully complete the WRITE; programmed WAIT ##STR24## ##STR25## returns HIGH.

*(M) = master mode, (S) = slave mode, I = input, O = output

Detailed Description Paragraph Table (10):

OPERATOR DESCRIPTION

+ plus - minus * multiply / divide .about. one's complement & bitwise AND .vertline. bitwise OR bitwise exclusive OR int convert to INTEGER type log use base Ex. a log b identifies a as a base b number exp raise to the power of Ex. a exp b is a to the b power fix convert to FIXED type && logical AND .parallel. logical OR ! logical NOT == EQUAL TO != NOT EQUAL TO < less than > greater than <= less than or equal to >= greater than or equal to >> right shift Ex. a >> b is "right shift a by b bits" << left shift Ex. a << b is "left shift a by b bits"

Detailed Description Paragraph Table (12):

asmblock name [{[%paremeter[=expression][, %parameter[=expression]. . .]]}] [{[in[, in ...]]; [out[, out ,,.]]}]

Detailed Description Paragraph Table (13):

symbol[micro]identifier= expression[, [micro] identifier=expression ...]; or symbol[micro]string= expression[, [micro] string=expression ...];

Detailed Description Paragraph Table (14):

variable[micro][type]identifier[[size]][=expression][,
[micro][type]identifier[[size]][= expression] ...];

Detailed Description Paragraph Table (21):

INSTRUC- TION OPERAND DESCRIPTION

lbsj source Load BS and jump. Load the program counter plus one and the condi- tion codes into the BS register and jump to operand. ldcc source Load condition codes. Replace 4 bits of condition code register with 4 bits of operand (CF, OF, SF, ZF). xld source Load parallel port input register with contents of externally addressed memory location. The operand specifies an external address in the range of 0 through 64K. NOTE: This instruction alters the value in the A register. The state of the CF, SF, and ZF flags is unknown after this instruction.

Detailed Description Paragraph Table (28):

A eor jmp MG seqblock upsample adc exp jne MGI sta variable add F jov MH stb verify alias fix jsi MHI stbs virtual and fixed jwf micro std wire asl gpio jze ML stf WS asmblock hex L MLI stl X asr init label mpy stmg xld B input lbsj nop stmgi xor begin int lda not stmh Y block integ ldb ora stmhi BS integer ldcc org stml callblock jcc ldd output stmli clc jcs ldf param stws cmp jeq ldl phantom stx com- jge ldws port sty puteline D jgt ldx real sub djne jle ldy rol subc down- jlf log ror subrblock sample duration jlt mac rts symbol end jmf manblock sec timezone

CLAIMS:

1. Apparatus for providing program code for a real time signal processor means having a memory means, and for concurrently generating memory access code for use by a host processor means so that the host processor means can change the contents of the memory of the signal processor means, wherein object code for the signal processor means and the host processor means are separately compiled by a respective signal processor compiler and a host processor compiler, and the signal processor means and the host processor means each include means for interfacing with each other, said apparatus comprising:

means for describing a block diagram representing a processing task for the signal processor means with a plurality of high level signal processing functional block means and a plurality of connections between said high level signal processing functional block means, each functional block means having a name, at least one of said functional block means having a parameter, and at least one of said functional block means having an indication of being accessible by the host processor means;

signal processor cell library means containing a plurality of source code blocks, each source code block representing at least a portion of a corresponding high level signal processing functional block means, at least one of said source code blocks containing a named variable for receiving a value for said parameter; and

signal processor compiler means coupled to said means for describing and to said signal processor cell library means, for analyzing said block diagrm, for obtaining code blocks from said signal processor cell library means needed to implement said block diagram, for associating said named variable with said parameter, and for compiling said code blocks to provide program code and data code for the memory means of the signal processing means, and a correspondence table associating at least one signal processor means memory location with said named cell variable, said correspondence table being for the host processor means such that said correspondence table of a translation thereof is used by the host processor compiler in the compiling of the object code for the host processor means, and the object code for the host processor means causes the host processor means to access the memory location of the signal processor means via the interface means of the host processor means and the signal processor means so that the host processor means can change said parameter value,

wherein said program code and data code will cause the signal processor means to

implement said processing task for the signal processor means, with said program code controlling the functioning of the real time signal processor means, and said data code providing initial parameter values for the signal processor means and initial sample data values for memory locations in the memory means of the signal processor means, and wherein said parameter values represent values of parameters of said functional block means.

4. Apparatus according to claim 1, wherein:

said means for describing a block diagram comprises means for representing a processing task for the host processor means with a plurality of high level host processor functional block means and a plurality of connections between said plurality of high level host processor functional block means, each host processor functional block means having a name, wherein said means for describing further comprises means for distinguishing between portions of said block diagram intended for compilation by the signal processor means and portions of said block diagram intended for compilation by the host processor means.

5. Apparatus according to claim 1, further comprising:

means for generating from said program code and said data code signal processor means boot code suitable for compilation by the compiler of the host processor means, wherein the host processor means can use said boot code to boot up the signal processor means.

6. Apparatus according to claim 1, further comprising:

correspondence table translation means for translating said correspondence table generated by the signal processor compiler means into code suitable for compilation by the compiler for the host processor means.

7. Apparatus according to claim 5, further comprising:

correspondence table translation means for translating said correspondence table generated by said signal processor compiler means into code suitable for compilation by the compiler for the host processor means.

10. Apparatus according to claim 9, wherein:

said signal processor means further comprises a parallel host port coupled to said program memory means and to said multiported central memory means, wherein said host port comprises the interface to the host processor means.

14. Method for providing program code for a real time signal processor means having a memory means, and for concurrently generating memory access code for use by a host processor means so that said host processor means can change the contents of the memory of said signal processor means, wherein object code for said signal processor means and said host processor means are separately compiled, and said signal processor means and said host processor means each include means for interfacing with each other, said method comprising:

describing and representing a processing task for said signal processor means in a block diagram with a plurality of high level signal processing functional block means and a plurality of connections between said high level signal processing functional block means, each functional block means having a name, at least one of said functional block means having a parameter, and at least one of said functional block means having an indication of being accessible by the host processor means;

providing a signal processor cell library means containing a plurality of source code blocks, each source code block representing at least a portion of a corresponding high level signal processing functional block means, at least one of said source code blocks containing a named variable for receiving a value for said parameter; and

analyzing said block digram in order to obtain needed code blocks from said signal

processor cell library means for implementing said block diagram;

associating said named variable with said parameter, and

compiling said code blocks to provide program code and data code for the memory means of the signal processing means, and a correspondence table associating at least one signal processor memory location with said named cell variable,

providing said correspondence table or a translation thereof for compilation as object code for said host processor means, wherein said correspondence table permits said host processor means to change a parameter value stored as a variable in said memory of said signal processor means.

16. A method according to claim 14, further comprising:

generating from said program code and said data code signal processor means boot code suitable for compilation by said compiler of said host processor or for direct storage in said host processor.

17. A method according to claim 14, further comprising:

compiling said correspondence table or a translation thereof in said host processor compiler;

18. A method according to claim 16, further comprising:

compiling said correspondence table or a translation thereof in said host processor compiler;

storing said boot code in said host processor; and

booting up said signal processor by providing said boot code stored in said host processor to said signal processor.

19. An apparatus for coupling a host processor and a programmable signal processor having a memory for storing a program and data so that the host processor has intelligent access to the signal processor memory, said apparatus comprising:

a) high level programming means for defining signal processor functions, said programming means including means for symbolically indicating host processor access to signal processor functions;

b) signal processor program compiling means coupled to said programming means for generating signal processor program code which causes said signal processor to implement said signal processor functions and for generating memory access code for the host processor, said memory access code including a correspondence table correlating symbolic indications indicated in said programming means with memory addresses in the signal processor memory;

c) host processor compiling means for generating host processor program code including reference to at least one of said memory addresses in said correspondence table, said host processor compiling means including means for receiving at least a portion of said memory access code; and

d) interface means coupling the host processor and the signal processor memory,

wherein said host processor program code causes the host processor to access the signal processor memory according to said symbolic indications indicated in said programming means.

25. Apparatus for defining access of a host processor which is interfaced with a programmable signal processor to permit the host processor to partially control the programmable signal processor, where the object codes of the host processor and the signal processor are separately compiled, and the programmable signal processor has a memory for storing the signal processor object code and data, said apparatus

comprising:

a) high level programming means for defining processing tasks for the signal processor, said programming means including means for indicating host processor access to at least a portion of one of said processing tasks; and

b) signal processor program compiling means coupled to said high level programing means for generating the signal processor object code which implements said processing tasks, and for generating a memory address list for the host processor, said memory address list indicating memory addresses corresponding to the host processor access to said tasks indicated by said high level programming means, said memory address list being read or received by a host processor compiling means.

WEST

Generate Collection Print

L5: Entry 60 of 235 File: USPT Jul 29, 2003

US-PAT-NO: 6601049
DOCUMENT-IDENTIFIER: US 6601049 B1

TITLE: Self-adjusting multi-layer neural network architectures and methods therefor

DATE-ISSUED: July 29, 2003

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Cooper; David L.	Fairfax	VA	22033	

APPL-NO: 09/ 578701 [PALM]
DATE FILED: May 26, 2000

PARENT-CASE:
This Application is a Continuation-In-Part of U.S. patent application Ser. No. 09/240,052, filed Jan. 29, 1999, now U.S. Pat. No. 6,490,571, which is a Continuation-In-Part of U.S. patent application Ser. No. 08/713,470, filed Sep. 13, 1996, now issued as U.S. Pat. No. 6,009,418, which claims the benefit of U.S. Provisional Patent Application Serial No. 60/016,707 filed May 2, 1996. The entire disclosures of these applications, including references incorporated therein, are incorporated herein by reference.

INT-CL: [07] G06 N 3/02

US-CL-ISSUED: 706/15; 706/2, 706/6, 706/38, 700/48, 700/49, 700/50
US-CL-CURRENT: 706/15; 700/48, 700/49, 700/50, 706/2, 706/38, 706/6

FIELD-OF-SEARCH: 706/2, 706/6, 706/15, 706/38, 706/41, 700/48, 700/49, 700/50, 700/17, 342/90, 342/92, 365/49, 382/115, 382/225, 382/224, 359/29

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

Search Selected Search ALL

	PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
☐	4591980	May 1986	Huberman et al.	712/16
☐	4977540	December 1990	Goodwin et al.	365/49
☐	5091864	February 1992	Baji et al.	706/42
☐	5151969	September 1992	Petsche	706/31
☐	5153923	October 1992	Matsuba et al.	382/158
☐	5214715	May 1993	Carpenter et al.	382/157
☐	5497253	March 1996	Stoll et al.	359/29
☐	5504487	April 1996	Tucker	342/90
☐	5636326	June 1997	Stork et al.	706/25
☐	6009418	December 1999	Cooper	706/15
☐	6052485	April 2000	Nelson et al.	382/225
☐	6490571	December 2002	Cooper	706/15

OTHER PUBLICATIONS

US 2002/0016782 A1, Feb. 7, 2002, Cooper, David L., Method and Apparatus for Fractal communication.

ART-UNIT: 2121

PRIMARY-EXAMINER: Patel; Ramesh

ATTY-AGENT-FIRM: Greenberb Traurig Kurtz, II; Richard E.

ABSTRACT:

A method and apparatus for using a neural network to process information includes multiple nodes arrayed in multiple layers for transforming input arrays from prior layers or the environment into output arrays for subsequent layers or output devices. Learning rules based on reinforcement are applied. Interconnections between nodes are provided in a manner whereby the number and structure of the interconnections are self-adjusted by the learning rules during learning. At least one of the layers is used as a processing layer, and multiple lateral inputs to each node of each processing layer are used to retrieve information. The invention provides rapid, unsupervised processing of complex data sets, such as imagery or continuous human speech, and captures successful processing or pattern classification constellations for implementation in other networks. The invention includes application-specific self-adjusting multi-layer architectures that employ reinforcement learning rules to create updated data arrays for computation.

36 Claims, 19 Drawing figures
Exemplary Claim Number: 1
Number of Drawing Sheets: 17

BRIEF SUMMARY:

1 BACKGROUND OF THE INVENTION

2 1. Field of the Invention

3 The invention relates in general to the field of neural networks, and in particular to a neural network architecture and method which utilizes self-adjusting connections between nodes to process information.

4 2. Related Art

5 Neural networks have been known and used in the prior art in computer applications which require complex and/or extensive processing. Such applications include, e.g., pattern recognition and image and voice processing. In these applications, neural networks have been known to provide greatly increased processing power and speed over conventional computer architectures. Several approaches to neural networking exist and can be distinguished from one another by their different architectures. Specifically, the approaches of the prior art can be distinguished by the numbers of layers and the interconnections within and between them, the learning rules applied to each node in the network, and whether or not the architecture is capable of supervised or unsupervised learning.

6 A neural network is said to be "supervised" if it requires a formal training phase where output values are "clamped" to a training set. In other words, such networks require a "teacher," something not necessarily found in nature. Unsupervised networks are desirable precisely for this reason. They are capable of processing data without requiring a preset training set or discrete training phase. Biological neural networks are unsupervised, and any attempt to emulate them should aspire to this capability. Of the following approaches, the Boltzmann/Cauchy and Hidden Markov models are supervised networks and the remainder are unsupervised networks.

7 At least eight principal types of feedback systems, also called backpropagation models, have been identified in the prior art. The Additive Grossberg model uses one layer with lateral inhibitions. The learning rule is based on a sigmoid curve and updates using a steepest ascent calculation. The Shunting Grossberg is similar, with an added gain control feature to control learning rates. Adaptive Resonance Theory models use two layers, with on-center/off-surround lateral feedback and sigmoid learning curves. The Discrete Autocorrelator model uses a single layer, recurrent lateral feedback, and a step function learning curve. The Continuous Hopfield model uses a single layer, recurrent lateral feedback, and a sigmoid learning curve. Bi-Directional Associative Memory uses two layers, with each element in the first connected to each layer in the second, and a ramp learning curve. Adaptive Bi-Directional Associative Memory uses two layers, each element in the first connected to each in the second, and the Cohen-Grossberg memory function. This also exists in a competitive version. Finally, the Temporal Associative Memory uses two layers, with each element in the first connected to each element in the second, and an exponential step learning function.

8 At least eight principal types of feedforward systems have been identified. The Learning Matrix uses two layers, with each element in the first connected to each element in the second, and a modified step learning function. Drive-Reinforcement uses two layers, with each element in the first connected to each in the second, and a ramp learning function. The Sparse Distributed Memory model uses three layers, with random connections from the first to the second layer, and a step learning function. Linear Associative Memory models use two layers, with each element in the first layer connected to each element in the second, and a matrix outer product to calculate learning updates. The Optimal Linear Associative Memory model uses a single layer, with each element connected to each of the others, and a matrix pseudo-inverse learning function. Fuzzy Associative Memory uses two layers, with each element in the first connected to each element in the second, and a step learning function. This particular model can only store one pair of correlates at a time. The Learning Vector Quantizer uses two layers, with each element in the first connected to each in the second, negative lateral connections from each element in the second layer with all the others in the second layer, and positive feedback from each second layer element to itself. This model uses a modified step learning curve, which varies as the inverse of time. The Counterpropagation model uses three layers, with each element in the first connected to each in the second, each element in the second connected to each in the third, and negative lateral connections from each element in the second layer to each of

the rest, with positive feedback from each element in the second layer to itself. This also uses a learning curve varying inversely with time.

9 Boltzmann/Cauchy models use random distributions for the learning curve. The use of random distributions to affect learning is advantageous because use of the distributions permits emulation of complex statistical ensembles. Thus, imposing the distributions imposes behavioral characteristics which arise from complex systems the model networks are intended to emulate. However, the Boltzmann/Cauchy networks are capable only of supervised learning. And, these models have proven to be undesirably slow in many applications.

10 Hidden Markov models rely on a hybrid architecture, generally of feedforward elements and a recurrent network sub-component, all in parallel. These typically have three layers, but certain embodiments have had as many as five. A fairly typical example employs three layers, a softmax learning rule (i.e., the Boltzmann distribution) and a gradient descent algorithm. Other examples use a three-layer hybrid architecture and a gamma memory function, rather than the usual mixed Gaussian. The gamma distribution is convenient in Bayesian analysis, also common to neural network research, and is the continuous version of the negative-binomial distribution. However, the underlying process for this model is a stationary one. That is, the probability distribution is the same at time t and time t+.DELTA.t for all .DELTA.t.

11 Studies of language change and studies of visual and acoustic processing in mammals have been used in the prior art to identify the mechanisms of neural processing for purposes of creating neural network architectures. For example, it has been noted that mammalian visual processing seems to be accomplished by feed-forward mechanisms which amplify successes. Such processing has been modeled by calculating Gaussian expectations and by using measures of mutual information in noisy networks. It has further been noted that such models provide self-organizing feature-detectors.

12 Similarly, it has been noted in the prior art that acoustic processing in mammals, particularly bats, proceeds in parallel columns of neurons, where feed-forward mechanisms and the separation and convergence of the signal produce sophisticated, topically organized feature detectors.

13 SUMMARY OF THE INVENTION

14 The invention in its preferred embodiment provides a method and apparatus for using a neural network to process information wherein multiple nodes are arrayed in multiple layers for transforming input arrays from prior layers or the environment into output arrays for subsequent layers or output devices. Learning rules based for reinforcing successful matches to templates, and simultaneously suppressing unsuccessful matches, are applied. Interconnections between nodes are provided in a manner whereby the number and structure of the interconnections are self-adjusted by the learning rules during learning. At least one of the layers is used as a processing layer, and multiple lateral inputs to each node of each processing layer are used to retrieve information.

15 The invention provides rapid, unsupervised processing of complex data sets, such as imagery or continuous human speech, and a means to capture successful processing or pattern classification constellations for implementation in other networks. The invention includes application-specific self-adjusting multi-layer architectures that employ template-based learning rules to alter and annotate data arrays. Such application-specific architectures include a textual or oral language parser, a basic file searcher, an advanced file searcher, an advanced file searcher that can search for propositions, a translator, a basic "smart" scanner, an advanced "smart" scanner or oral parser, and a dialect parser for oral language. These applications have a number of common features. Inputs are delivered into a flexible number of channels, determined by the total number of scanned letters, input phonemes, words, or propositional values in the input sample. The arrays in each channel are then combined in patterns that depend on values derived from lookup tables or templates. Feedback from one or two layers higher in the central processing

segment of the architecture further alters the arrays, where the learning rules reinforce template matches and also decrement failures to match. The lookup or template-matching steps also alter the arrays to prevent confusion of the data and to augment the information carried forward through the architecture. The output channels transform the arrays into final form as required, as well as reset the initial processing weights for the next processing cycle. These outputs take whatever format desired; they may be printed text, statistical information in digital or graphic form, oral outputs through speakers, data in a register, or inputs to other processing applications.

DRAWING DESCRIPTION:

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, features, and advantages of the invention will be apparent from the following more particular description of preferred embodiments as illustrated in the accompanying drawings, in which reference characters refer to the same parts throughout the various views. The drawings are not necessarily to scale, emphasis instead being placed upon illustrating principles of the invention.

FIG. 1 illustrates a diagrammatic view of the self-adjusting layer architecture of the invention according to a preferred embodiment.

FIG. 1a illustrates a diagrammatic view of the self-adjusting layer architecture of the invention according to an embodiment wherein the invention is used to program a second neural network.

FIG. 2 illustrates a side view of a single channel.

FIG. 3 illustrates details of internal feedback in central processing layers.

FIG. 4 illustrates the effect of"leaky" processing.

FIG. 5 shows multiple "leaky" inputs merging at the central processing layers, and outputs proceeding to different implementation channels.

FIG. 6 depicts the front vowels for the participants in the Northern Cities Shift.

FIG. 7 illustrates the first three iterations in the generation of a Cantor set.

FIGS. 8-10 show graphical illustrations drawn from natural language studies.

FIG. 11 shows a schematic design for a textual or oral language parser.

FIG. 12 shows a schematic design for a basic file searcher.

FIG. 13 shows a schematic design for an advanced file searcher.

FIG. 14 shows a schematic design for an advanced file searcher that can search for propositions.

FIG. 15 shows a schematic design for a translator.

FIG. 16 shows a schematic design for a basic "smart" scanner.

FIG. 17 shows a schematic design for an advanced "smart" scanner or oral parser.

FIG. 18 shows a schematic design for a dialect parser for oral language.

DETAILED DESCRIPTION:

1 DETAILED DESCRIPTION OF THE DRAWINGS

2 By studying language acquisition and evolution, one can identify learning curves in language acquisition. These learning curves can be characterized by negative-binomial statistics. One can also identify patterns of change which can be explained by underlying attractors with fractal dimension. That is, the dimension of the sets is not the topological dimension--1 for a line, 2 for a plane, etc.--but a related measure of the size and complexity of the set, whose measure can be fractional.

3 The measures of these attractors are consistent with Cantor sets in three dimensions. In Cooper, "Linguistic Attractors: The Cognitive Dynamics of Language Acquisition and Change," chapters 1-3, which is incorporated by reference herein, it is demonstrated that Cantor sets are sufficient (though not necessary) to establish a universal Turing machine. That is, they are sufficient to demonstrate an ability to compute any function. This reference also shows a correlate of these attractors with statistical patterns derived from children learning English, which provided the evidence for a learning process which converges on a negative-binomial distribution.

4 The self-adjusting layer architecture of the invention is motivated by the above and other considerations from human language acquisition and change, as well as by the general constraints posed by the structure of the human perceptive apparatus and systems for muscle control. It is based on multiple layers; with one or more layers devoted to processing input information transmitted along "leaky" channels; at least one processing layer; one or more layers devoted to processing outputs; feedback from the outputs back to the processing layer; and inputs from parallel channels, also to the processing layer. With the exception of the feedback loop and central processing layers, the network is feedforward. The learning rules reinforce successful matches to stored or learned patterns, again derived from considerations of human language acquisition.

5 The self-adjusting layer architecture of the invention according to a preferred embodiment is derived from a model of human speech acquisition based on random pattern matching. The requirement for negative reinforcement in human speech acquisition is controversial, but the consensus view seems to be that it is not required. A stochastic process relying on positive reinforcement which also yields the learning curves observed in human subjects as they learn to speak is consequently preferred for the matching process. Bose-Einstein statistics capture such a process, which can also be modeled as a Polya process that models sampling with replacement from sample populations. This process will also lead to a fractal pattern of synapse sequences corresponding to each learned pattern. Such a pattern is also to be expected from data on human language change.

6 In the limit, Bose-Einstein systems or Polya processes converge to learning curves with negative-binomial distributions, but the fundamental process is a non-stationary one. As a further consequence, any model which relies on stationary processes (i.e., any Markov process, normal processes, Bernoulli processes, gamma processes, etc.) is only an approximation to this model, and accurate only in the limit as nodes approach infinity.

7 The architecture of the self-adjusting layer architecture at once mimics the input and output processing found in human perceptions and muscle control, and permits a variant of recurrent network architectures to learn in an unsupervised environment. Parallel "leaky" channels combine the strengths of Markov or other hybrid architectures with a more structured higher scale architecture. That is, major formations in the human brain most probably are joined on a large scale by pattern instructions, but internally by random connections. The multiple layers within a channel reflect the fact that the human nervous system does considerable processing before information is presented to the brain, and, again, considerable processing after leaving the brain before impulses arrive at the muscles. This not only contributes a time delay before feedback but also additional content to the commands to the muscles--commands which must have been learned.

8 FIG. 1 illustrates a diagrammatic view of the self-adjusting layer architecture of the invention according to a preferred embodiment. The diagram depicts typical elements in the network, but each layer in a channel could have many more than three elements per layer. These will be connected randomly to elements in the next layer, so it is not necessarily the case that each element in one layer is connected to each in the next. More than three layers may be necessary before arrival at the processing level, and more than three layers may be necessary before reaching the implementation layer (e.g., muscles).

9 In FIG. 1, information arrives from the left to a perceptual apparatus, which encodes the data, and passes it to the next layer. This flow continues to the processing layer, where inputs converge with inputs from parallel channels and output from a feedback loop connected to the primary channel.

10 The architecture uses feedforward nodes, randomly linked to higher levels within prescribed channels to process the inputs. Although not depicted, connections from each layer to the next are random: each node is not necessarily connected to each one in the layer above it. There is at least one central layer with lateral connections to other channels. This will be called the lateral processing layer.

11 Outputs proceed to the right, again with random feed-forward connections. Feedback from the output channels are merged with inputs to the lateral processing layer. This feedback is also merged with feedback from the lateral processing layer and from the initial output layer. This feedback is not depicted in FIG. 1, but is depicted by rings in FIGS. 2 and 3, discussed below. Outputs from other channels are also presented to the lateral processing layer.

12 Central processing is actually embedded between input and output channels, so that there is no discrete transition. A side view of a single channel is at FIG. 2, and details of internal feedback in the central processing layers is at FIG. 3. The rationale for these feedback and processing arrangements is set forth in the discussion of the learning rule for the network below.

13 The FIGS. depict a seven-layer architecture, but the actual number of layers would be dependent on input pre-processing, and output processing. This could require as few as one input layer or more than three feed-forward input layers, as well as at least one and possibly more than three feed-forward output layers in their respective channels. The essential result is, however, that structured data is presented to the central processing layers from input channels, and that outputs are further processed as required for final implementation. In natural language, for example, this would mean the network would take acoustic inputs, encode them in the input channels, process them in the central processing layers, and then process out-going motor signals in the necessary channels to a number of different muscle groups.

14 The three-layer central processing core of the architecture will produce the necessary computational characteristics.

15 It should be noted that in FIG. 2 the final input layer is also the first central processing layer. Similarly, the final processing layer is the initial feed-forward layer to output channels.

16 FIG. 3 shows the internal feedback within the central processing layers. External feedback from the beginning and the end of the output channels is merged with the inputs at the initial central processing/final input layer. Other channels feed into the lateral processing layer. Lateral connections in the central layer are crucial to a multi-channel architecture. An additional layer with lateral connections is optional, but makes processing of sequences easier for the network. Consequently, for language processing, where sequences must be processed with minimal load on memory, the additional lateral connections may be necessary.

17 In a complex organism, channels would be linked to specific perceptual organs

and to output musculature keyed to specific behavior. Success is reinforced. This is a mild form of a competitive network, so that information will be organized topologically when processed, but the topology will not be strict. The random connections will make the upward flow of information within the channel "leaky." The effect of"leaky" processing is diagramed in FIG. 4.

18 FIG. 5 shows multiple "leaky" inputs merging at the central processing layers, and outputs proceeding to different implementation channels. Processing in the central layers would be affected by the structure of lateral channels as well. These can be designed into the architecture, or evolve through the same rules as apply to the vertical channels.

19 Feedback can be processed by any of several expectation maximization schemes. Such schemes are described, e.g., in Linsker, "Neural Information Processing Systems 5" Morgan Kaufmann Publishers, 1993, pages 953-960, the disclosure of which is incorporated herein by reference. However, any reinforcement over the critical threshold must update the successful interconnections according to Bose-Einstein statistics (originally, equiprobability assigned to different constellations of indistinguishable particles). This can be done as follows.

20 Begin with a Polya urn scheme, with m.sub.0 balls in the urn, m black and m.sub.0 -m red. For each drawing, replace each black ball with two black balls, and each red ball with two red balls (sampling with replacement). Then for n.sub.0 drawings, of which n balls were black, and n.sub.0 -n were red, we have the Polya distribution for the probability that the next ball will be black ##EQU1##

21 This is also the Bose-Einstein distribution for m.sub.0 cells (neurons/synapses) and n.sub.0 particles (inputs), with a constellation of m cells (neurons/synapses) with a total of n indistinguishable particles (successes). A constellation of m cells with a total of n indistinguishable balls for n.sub.0.fwdarw..infin., m.sub.0.fwdarw..infin., such that n.sub.0 /m.sub.0 =p, p finite is then ##EQU2##

22 This is a special case of the negative-binomial distribution, as required from learning curve data on human speech acquisition.

23 The network requirement is then n.sub.0 reinforced pathways across m.sub.0 synapses, both large, with n.sub.0 /m.sub.0 =p. When the Polya process has the substitution factor c.gtoreq.0, the transition probability from E.sub.n (n black) to E.sub.n+1 (n+1 black) is ##EQU3##

24 This leads to a special form for a rule to update synaptic weights. In general, Hebbian learning rules are expressed in the form

.DELTA.w.sub.kj (n)=F(y.sub.k (n), x.sub.j (n))

25 for synaptic weights from j to k, where F is a function of pre- and post-synaptic activity. Frequently, models use the following special case

.DELTA.w.sub.kj (n)=.eta.y.sub.k (n)x.sub.j (n)

26 where .eta. is a positive learning rate parameter. To avoid saturation of the weights, this is often modified by the addition of a forgetting factor:

.DELTA.w.sub.kj (n)=.eta.y.sub.k (n)x.sub.j (n)-.alpha.y.sub.k (n)w.sub.kj (n)

27 The rule has also been expressed in terms of the covariance between pre- and post-synaptic activities:

.DELTA.w.sub.k (n)=.eta.E[(y.sub.k (n)-y.sub.k (x.sub.j (n)-x.sub.j)]

28 where E is the expectation for the activity.

29 The update rule for the self-adjusting layer architecture uses the learning

rate parameter in a very different way. The form of the rule with a decay rate (to avoid saturation of the weights) is ##EQU4## .eta.positive; .delta.(t).gtoreq.0 and monotone increasing

30 where r/s is the synaptic weight at epoch n-1, and .vertline.r/s.vertline..ltoreq.1. Here r is increased for excitation, and decreased for inhibition. It only changes when there is activity at the synapse. The decay rate .delta.(t) varies with time, and does not depend on synaptic activity. The cumulative update across all inputs j would then take the form ##EQU5##

31 where .alpha.(n)+.beta.(n)=.eta.(n), all positive

32 where .alpha.(n) and .beta.(n) count the numbers of excitatory and inhibitory inputs, respectively.

33 We can apply the Cohen-Grossberg theorem to this rule immediately to demonstrate that the Self-adjusting layer architecture rule is capable of content-addressable memory. That is, any input of a portion of the memory can function to recall the rest. Briefly, the theorem states that for the class of neural networks described by the following system of coupled nonlinear differential equations: ##EQU6##

34 the class of neural networks admits a Liapunov function, defined as ##EQU7##

35 where ##EQU8##

36 When the synaptic weights of the network are "symmetric":

C.sub.ij =C.sub.ji

37 the function A.sub.j,(u.sub.j) satisfies the condition for "nonnegativity":

a.sub.j (u.sub.j).gtoreq.0

38 and the nonlinear input-output function .phi..sub.j (u.sub.j) satisfies the condition for "monotonicity": ##EQU9##

39 then the Liapunov function E of the system defined by Eq. (2) satisfies the condition and the system is globally asymptotically stable. That is, the system will be capable of content addressable memory.

40 Applying the Cohen-Grossberg theorem to the self-adjusting layer architecture, we note immediately that pre- and post-synaptic weights are identical, and thus, use symmetric weights between nodes. Then, if we set a.sub.j (u.sub.j)=1, it remains to show that the function .phi.'(u.sub.j).gtoreq.0. This is the case for .alpha.(n).gtoreq..beta.(n) and .delta.(t) sufficiently small. Thus, we have monotonicity, and the self-adjusting layer architecture is capable of content addressable memory. It is also capable of the other attributes necessary for computation.

41 The learning rule for the self-adjusting layer architecture naturally leads to negative-binomial learning curves for the network--comparable to curves observed in humans acquiring natural language. This is an emergent property of the rule and need not be imposed by a structured learning regime as in the Boltzmann and Cauchy networks. The rule will lead to clusters of nodes in each layer associated with each successful multi-channel pattern. This is an important characteristic which will permit the network to function as a Turing machine. This can be demonstrated by considering these clusters of favored nodes as attractors. Their measurement is consistent with natural language in humans, and indicative of structured sets capable of computation. For example, the most useful measure we can apply to phonological attractors is Hausdorff dimension, which permits us to classify them as fractal sets. To make this application, we must establish a measure over our phase space. This is simple enough for formant frequencies, and can be extended to axes that code for other

relevant factors, such as source, city, generation, gender, and conditioning environment. Using the usual Euclidean metric, we can establish a "distance" and can measure a "diameter." Then, we can say the following.

42 The diameter of a set U is .vertline.U.vertline.=sup{.vertline.x-y.vertline.: x,y.epsilon.U}. For E.OR right..orgate.U.sub.i and 0<.vertline.U.sub.i.vertline..ltoreq..delta. for each I, {U.sub.i } is a .delta.-cover of E. For .delta.>0, s.gtoreq.0, ##EQU10##

43 where the infimum is over all countable .delta.-covers {U.sub.i } of E. H.sub..delta..sup.s is an outer measure on semantic space. The Hausdorff s-dimensional outer measure of E is ##EQU11##

44 This limit may be infinite, since H.sub..delta..sup.s increases as .delta. decreases. H.sup.s (E) is a metric outer measure since it satisfies a triangle inequality. The restriction of H.sup.s to the .sigma.-field of H.sup.s -measurable sets is called Hausdorff s-dimensional measure.

45 For any E, H.sup.s (E) is non-increasing as s increases from 0 to .infin.. For s<t,

H.sub..delta..sup.s (E).gtoreq..delta..sup.s-t H.sub..delta..sup.t (E)

46 This implies that if H.sup.t (E) is positive, H.sup.s (E) is infinite, which further implies that there is a unique value, dim E, called the Hausdorff dimension of E, such that

H.sup.s (E)=.infin. if 0.ltoreq.s.ltoreq.dim E; H.sup.s (E)=0 if dim E<s<.infin.

47 Ascribing probabilities to phonological attractors is therefore meaningful. We can ascribe fractal dimension to them as well.

48 Unfortunately, direct calculation of the Hausdorff dimension is often intractable. It can be approximated with another function, called the correlation function C(r), for varying distances r, which is given by ##EQU12##

49 Where N is the number of data points, r is the distance used to probe the structure of the attractor, X.sub.i and X.sub.j are pairs of points, and .theta. is the Heaviside function, where .theta.(x)=1 for x>0 and .theta.(x)=0 otherwise.

50 We estimate the dimension of the attractor by taking the slope of the linear portions of the curves. Since generally InC(r).about.dln(r), the slope will converge to d when the space is saturated. That is, the minimum number of variables to describe the manifold in n-space occupied by an attractor will be d.

51 FIG. 6 depicts the front vowels for the participants in the Northern Cities Shift. In the figure, the monophthongs, /i/, /e/, and particularly /.ae butted./, show substantial linear segments. These are all participants in the chain shifts revealed in a detailed study of dialect change among the white populations of cities ranging from upstate New York across Minnesota, while the diphthongs generally have not. The diphthong signature on this chart is typical of sounds which are generally stable. The front monophthongs have dimensions ranging from 1.4 to 1.6.

52 It will now be shown that the structure of these set hierarchies allows the communication and processing of information. To see this, consider phonological attractors, which required several axes apiece to describe them fully, with each projection on each axis with dimension less than one. With fractal attractors, the projection on almost all dimensions in semantic space will in fact be zero. For those dimensions along which the dimension is greater than zero and less than one, it is possible to construct a fractal set, called a cascade, which has the characteristics of a Turing machine.

53 A simple example of such a cascade, used by Mandelbrot as a simple model for random errors in data transmission, is the Cantor set. This set consists of residues of the unit interval [0,1] from which the middle third is subtracted, and then the operation is applied to each successive residue ad infinitum. The resulting set has dimension s=0.6309 . . . and H.sup.s (E)=1.

54 Instead of this simple set, we can construct a more involved set, which I will call a Godel cascade, based on a generalized Cantor set. Specifically, we can define a finite union of closed intervals of the unit interval E.sub.0, E.sub.0.OR left.E.sub.1.OR left.E.sub.2 . . . . For each interval I of E.sub.j, we specify E.sub.j+1.andgate.I by selecting an integer m>2 and letting the subintervals J.sub.1, J.sub.2, . . . , J.sub.m be equally spaced, with lengths given by ##EQU13##

55 Then

m.vertline.J.sub. i.vertline.+(m-1)d=.vertline.I.vertline.(1.ltoreq.i.ltoreq.m)

56 where d is the spacing between two consecutive intervals J.sub.i. The value for m may vary over different intervals i in E.sub.j. The resulting set has dimension s and H.sup.s (E)=1.

57 Now, following Martin Davis in the study of recursive operations and computability, if we associate each of a set of symbols with a different odd integer, we can construct the unique Godel number for any expression ##EQU14##

58 where r is the Godel number, a.sub.k is the integer for the kth symbol and Pr(k) is the kth prime number.

59 To create a cascade analog to this procedure, if we set m=2 and

d.sup.-1 =Pr(k).sup.a.sup..sub.k

60 in the construction of a cascade we can then in principle recover a unique set of residues for any expression by measuring the gaps between the residues. For the expression illustrated in FIG. 7, extended to n symbols, the expression can be recovered in reverse order by comparing the first gap to the residue one level up, the second gap to the residue two levels up, and so on to the 2.sup.n th gap to the residue n+1 levels up.

61 This is merely the simplest procedure yielding this result. For equally spaced J.sub.m, for example, we can also set w=2m-1 such that

w=Pr(k).sup.a.sup..sub.k

62 The expression can then be recovered from the ratio between each i and J.sub.i derived from it. Alternatively, if we let m stand for w, the expression would be recoverable by counting the J.sub.i. Clearly, since Godel expressions can model any proposition in any formal system, cascades can, too. Other, more efficient cascades are n.sub.0 doubt possible as well.

63 A collection of such cascades would be a means to encode a collection of expressions, and hence would constitute a Turing machine.

64 Since Godel numbering is not the only means to reduce expressions to integers, this demonstration is only good in principle, but it shows that fractal dimensions make sense. They provide a sufficient (although not necessary) condition for the full gamut of cognitive functions.

65 The generalized Cantor set from which the Godel architecture and learning algorithm adds structured feedback, which allows correlations without a supervised learning situation, and a non-stationary update rule, which will yield the necessary fractal set components to match observations, since activation patterns in the central processing nodes will produce two

dimensional Cantor sets (with fractal dimension less than 2, and generally greater than 1 for high information sets). These sets can then be combined. With a recursive learning algorithm and the processing layers seeking local mutual information maxima, we have the three components necessary for a Turing machine: recursion, composition, and minimalization. Davis also demonstrated that the minimalization operation need be performed only once, which means that the lateral processing layer is sufficient to make the overall architecture function.

66 It is probably significant to note that PET scans of humans show dimensions generally between 1 and 2, as we would expect from this architecture.

67 FIG. 7 illustrates the first three iterations in the generation of a Cantor set. Cantor sets are formed by the iterated concentration of the set members into increasingly small subsets of the unit segment [0,1]. In the classical Cantor set, the middle third of the segment is removed at each iteration; that is, the middle third of the segment [0,1], then the middle third of the remaining two pieces, then the middle third of the remaining four pieces, and so on. In a generalized Cantor set, the size of the piece removed is variable. The capability to reconstruct the size of the removed pieces by examination of the residues provides the computational capacity for such sets.

68 FIGS. 8-10 are drawn from natural language studies, and show similar dimensions at all levels of abstraction. FIG. 8 shows the back vowels from the same Northern Cities Vowel Shift study as FIG. 6; FIG. 9 shows case (i.e., nominative, genitive, dative, accusative) in Old English; and, FIG. 10 shows mood (indicative, subjunctive, imperative, and infinitive) in Old High German. The correlation signatures are all similar, and the underlying dimension is nearly the same as well for the high content sets: approximately 1.2 for the Old English dative, and 1.4 for the Old High German subjunctive.

69 The present invention can be practiced using digital network emulation software on a general purpose computer, or using special-purpose neural networking hardware, or some combination thereof. As the learning rule depends on probability distributions, digital implementation can calculate the distributions directly, or they can model the process as one of synapse pathway recruitment. In the latter case, since synapses cannot increase without bound, the population of synapses must be renormalized at each step. This is most easily accomplished with a uniform decay rate in the strength of connections.

70 Similarly, hardware or software/hardware simulations are possible with quantum devices based on particles obeying Bose-Einstein statistics. This population of particles includes photons, and therefore laser devices, but n.sub.0 particles subject to Pauli exclusion, such as electrons.

71 A preferred implementation of the invention is by means of digital emulation of a network, with the program specifying a series of lattices, each node incorporating the update formula, and each node on a given lattice mapped to nodes on the next lattice by a random, Gaussian distribution. Feedback may similarly require a Gaussian connection between the feedback source and the target processing layer. For single or small numbers of parallel channels, the model can be implemented on a standard Intel-based platform, e.g., one having a Pentium processor running at 100 MHz, or on a Macintosh platform with similar computational power. Larger emulations may require a workstation-class computer.

72 Implementation of the invention is also possible by use of a specialized chip, where the internal circuits respond according to the update rule.

73 The fact that the network generates Bose-Einstein statistics, and does not need to have them imposed by program, means that hardware implementations are also possible using components exhibiting quantum behaviors, such as lattices of atoms which respond to given frequencies. Bose-Einstein statistics, for example, describe the Einstein radiation mechanism, where the critical parameters are the absorption and spontaneous emission of photons. For this

reason, Bose-Einstein statistics underlie physical devices such as lasers, which could thus be adapted to implementation of this type of network.

74 PROGRAMMING OF THE LEARNING RULES

75 The learning rules are based on a Polya process, which reinforces success. This is a non-stationary process which induces Bose-Einstein statistics when conducted across large numbers of synapses over many iterations. In the limit, as the iterations approach infinity, the resulting distribution converges to a negative-binomial distribution. The rules themselves may be implemented in three different variants: 1) a basic synapse model; 2) a competitive-learning version focused on neurons rather than synapses; and 3) a hybrid feedforward/back-propagation version in which feedback weights can employ a variety of maximization schemes. The network can run in synchronous mode for all three versions by setting a common clock for the network, with each epoch updated simultaneously across the network. The network can also run in asynchronous mode by setting a non-zero epoch time for each node, but updating weights only for the nodes involved when one fires. Since the asynchronous mode only uses local interactions, some optimization schemes in the hybrid version would not run in this mode.

76 The rules for the basic synapse model can be programmed as follows: a. Provide each neuron an address on a two-dimensional layer, similar to cells on a spreadsheet. b. Specify the connections between each layer, again, similar to specifying connections between cells on multiple spreadsheets in a workbook. c. For each neuron on layer j, we then have inputs from layer i and outputs to layer k. d. For each synapse linking i to j: If i fires, reset the original weight r/s to: ##EQU15## for learning constant c>0 For threshold .theta..sub.j for the neuron, when ##EQU16## j fires e. Calculate for all neurons on level j. f. Repeat calculations for all neurons on level k. g. Initial inputs to the first layer are measurements from the environment (signals from light sensors, low/high pass filters from sound equipment, etc.). h. Final outputs are to devices connected to the environment (printers, muscles, speakers, etc.)

77 The rules for competitive learning are similar, but modified as follows: a. Lateral connections (i.e., within the same layer) between neurons are specified as inhibitory. b. Step d above is modified as follows: If i fires, and if j fires, i.e., when ##EQU17## Then reset r/s as above.

78 In these rules, the feedback loops in the architecture are treated identically to the pathways from input to output. To employ alternate backpropagation maximization schemes, however, such as Linsker's "infomax," the feedback loops can be treated differently, with the weights reset according to the maximization scheme. This makes the network a hybrid feedforward/backpropagation network. This approach is more expensive in memory, but potentially faster.

79 To model "forgetting," which avoids saturation of the synaptic weights, any of these versions can be modified by introducing a decay factor. Since the decay factor is not related to activity, but simply to time, any synaptic junction that is inactive will eventually approach a connectivity of zero, for any non-zero decay rate. Decay at a synapse can then be modeled by: ##EQU18## for .alpha.+.beta.=.eta. all positive; .delta.(t).gtoreq.0 monotone increasing

80 where .alpha. is the number of excitations, and .beta. is the number of inhibitions.

81 Other decay rates are possible. For example, the common form ##EQU19##

82 The decay can also consist of a simple instruction to reset to default values.

83 PROGRAMMING THE ARCHITECTURE

84 The architecture has three components: 1) an input component; 2) a processing component; and 3) an output component. It therefore has a minimum of three

layers. When there are more than three layers, the components overlap. That is, the last input layer is also the first processing layer. Similarly, the last processing layer is also the first output layer.

85 The input component consists of one or more feedforward layers. The first layer takes its input from measurements of the environment as described above.

86 The processing layer consists of three or more layers. At least the middle layer has lateral connectivity within a given channel, and to other channels. Lateral connectivity within other layers of this component can be specified as desired. The layers also have internal feedback as indicated in the diagrams. Output feedback feeds to the first processing layer.

87 The output component has one or more feedforward layers, with final output to the environment as described above. Feedback from the final output goes to the first processing layer.

88 The architecture must have a channelized structure because the underlying Polya process converges to a stationary distribution as the number of actions goes to infinity. Hence the learning characteristics are maximized when the number of elements in each layer in a channel is the minimum necessary to process the data.

89 The "leaky" processing feature of the network can be induced by allowing random connections outside a given channel when the signal passes up the layers. This can be accomplished, for example, by employing a Gaussian distribution so that connections from one layer to the next are scattered around a mean corresponding to a direct vertical connection. The variance of the Gaussian distribution would determine how "leaky" the channel was.

90 Learning characteristics of the network are also affected by how channels are interconnected. If related data is processed in adjacent channels in the network, it would find the correlation much more quickly than if the channels were widely separated.

91 Consequently, the learning characteristics of the network are crucially linked to the size and interconnections of the channels. This is particularly so due to the simple nature of the learning rules.

92 The attractor concept underlying this design is also important because, as illustrated in FIG. 1a, successful combinations of weights and channels can be recorded and programmed in other networks as, for example, pattern classifiers or processors, so that the new networks need not be trained for that capability. In this way, recorded attractors can be used as a programming language for feedforward or hybrid neural networks. Specialized circuits or chips can also model the attractors.

93 The disclosed neural network architecture is particularly suited to neural network implementations which use specialized chips and very-large scale integrated (VLSI) circuit technology. In particular, floating gate transistor designs, such as that disclosed by Hasler, et al, in "Single Transistor Learning Synapses," Advances in Neural Information Processing Systems 7", MIT Press, the entire disclosure of which is incorporated herein by reference, can be applied at each node of the architecture of the invention. In such applications, the learning rules can be implemented by adding electrons (through injection) or removing electrons (through tunneling). On a larger scale, the architecture of the invention can be implemented using one or more Field Programmable Gate Array (FPGA) devices such as those disclosed by Martin Bolton in. "Programmable Arrays," The Electrical Engineering Handbook, 2d ed, the entire disclosure of which is incorporated herein by reference. Moreover, the network architecture of the invention also lends itself toward designing such FPGA devices by modeling the connections and contents of the logic cells that drive them. The network architecture of the invention can be implemented using other programmable logic devices (PLDs) as well.

94 FIGS. 11-18 show schematic diagrams illustrating the principles of application-specific self-adjusting, multi-channel, multi-layer architectures that employ non-stationary learning rules to create fractal dimensional data arrays for computation. In particular, the following applications are described: a textual or oral language parser (FIG. 11), a basic file searcher (FIG. 12), an advanced file searcher (FIG. 13), an advanced file searcher that can search for propositions (FIG. 14), a translator (FIG. 15), a basic "smart" scanner (FIG. 16), an advanced "smart" scanner or oral parser (FIG. 17), and a dialect parser for oral language (FIG. 18).

95 These applications have a number of common features. Inputs are delivered into a flexible number of channels, determined by the total number of scanned letters, input phonemes, words, or propositional values in the input sample. The arrays in each channel are then combined in patterns that depend on values derived from lookup tables or templates. Feedback from one or two layers higher in the central processing segment of the architecture further alters the arrays, where the learning rules reinforce template matches and also decrement failures to match. The lookup or template-matching steps also alter the arrays to prevent confusion of the data and to augment the information carried forward through the architecture. The output channels transform the arrays into final form as required, as well as reset the initial processing weights for the next processing cycle. These outputs take whatever format desired. For example, they may be printed text, statistical information in digital or graphic form, oral outputs through speakers, data in a register, or inputs to other processing applications.

96 Learning rules based on the Polya urn scheme described above would work in this architecture, as would any other convenient rules that reinforce success and suppress mismatches with templates. The effect of such rules is to merge and direct arrays into active channels, while reducing the activity in neighboring channels. The geometric pattern of this process parallels the iterative process used in constructing Cantor-like sets, and thereby creates data arrays across the layers of the architecture with fractal dimension. The actual content of the arrays can theoretically range from single digits to complex contents, as convenient. If restricted to a single bit of content per node, the information would be carried as combinations of active nodes and their geometric distribution, as in FIG. 7. With greater content, the physical distribution can be less important, and more content can be processed and stored at each node. In the applications in FIGS. 11-18, the input content is at least a byte per channel, and increases as the arrays are processed.

97 The parser in FIG. 11 receives oral or written input separated into words. Oral inputs received as features or formants would require the more complex architecture at FIG. 17. The words proceed to the next layer, where the network accesses a glossary or dictionary, which labels the words by type. Lateral feedback at this level would suppress impossible combinations. At the next layer, the network matches each adjacent pair of words to possible templates, such as Determiner+Noun. Feedback to the previous layer would suppress further impossible combinations, and reinforce matches. Lateral feedback would also do this. At the third processing layer, the network would access clausal templates against the remaining pairs of words, again suppressing impossibilities and reinforcing matches. This layer would be capable of checking for word-order clues. Feedback from this layer to the first processing layer can check for verb agreement and other such characteristics, thereby reducing the number of remaining options, and reinforcing the rest. At the fourth processing layer, the network accesses sentence templates, which allows it to check for textual relations to other sentences, and to correlate portions of compound, complex, conditional, and other sentence types. Finally, the network resets itself for the next sentence and produces propositional outputs and any desired statistical data. At this level, the parser would be capable of reducing the input text to its logical propositional components (e.g., `if a then b,` `either a or b,` `both a and b,` `a but not b,` etc.), as well as to basic syntactic components V(X, Y, Z)+A, where `V` is the verbal component of each clause, `X` the focus, `Y` the target (if any), `Z` the reference (if any), and `A` is the set of all adjuncts appended to each clause.

98 The file searcher designs in FIGS. 12-14 are related to the parser in two ways. First, they share the architectural features mentioned above, with more and more layers added as more and more advanced features are added. Second, they rely on parser outputs as templates for the search. In FIG. 12, the basic search design relies on X- or V-template inputs from the parser. The X-template would produce a noun index related to the focus (typically the subject) noun phrases in a basic text, thereby creating relatively sophisticated search parameters simply by parsing a base document of interest. Similarly, the V-template would employ the verbs as the basis for search. The more advanced search architecture at FIG. 13 allows more processing of the searched files before employing the parser inputs. The searcher would then match X- or V-templates from the base document against comparable arrays in the searched files, thereby creating an index of matching clausal parameters, rather than simple nouns or verbs. The most sophisticated of these searchers is at FIG. 14, which would match entire propositional outputs from the base document and all searched files.

99 FIG. 15 applies this approach to language translation. It takes the propositional output from a parsed document (e.g., from FIG. 11) or a parsed oral expression (e.g., from FIG. 17) in one language. It then looks up the words for matches in a second language, adjusts the alternatives against clausal templates of the second language, and finally against full sentence templates. The output would be an expression in the second language with the same propositional content as the input expression.

100 FIG. 16 represents a "smart" scanner, which takes inputs from a scanning device, matches the scanned characters against stored font templates, and adjusts the results based on a subsequent retrieval of dictionary information. As with the searcher design, smart scanners can employ any of the lookup/template levels from the parser architecture to adjust the final estimate of what has been scanned.

101 The most sophisticated scanner is represented at FIG. 17, which brings sentence templates to bear before producing an output. This architecture would also work for oral input, with phoneme data rather than character data.

102 Finally, FIG. 18 shows a design to help calibrate verbal signals. It takes oral inputs as processed into features or formants, then adjusts those estimates for the phonemic environment, for the estimated age and gender of the speaker, and then for regional (dialect) effects. The final output is in standard phonemes, which can be employed by the advanced parser at FIG. 17.

103 While the invention has been particularly shown and described with reference to a preferred embodiment thereof, it will be understood by those skilled in the art that various changes in form and details may be made therein without departing from the spirit and scope of the invention.

CLAIMS:

The embodiments of the invention in which an exclusive property or privilege is claimed are defined as follows:

1. A method for processing information, comprising the steps of: using a neural network to process said information, said step of using a neural network further comprising: using a plurality of nodes arrayed in a plurality of layers to transform input arrays from prior layers or the environment into output arrays with fractal dimension for subsequent layers or output devices; applying learning rules based on non-stationary statistical processes; using interconnections between nodes whereby the number and structure of said interconnections are self-adjusted by the learning rules during learning; using at least one of said plurality of layers as a processing layer; using a plurality of lateral inputs to each node of each of the said at least one processing layer to retrieve information.

2. The method according to claim 1, wherein said non-stationary statistical processes comprise a Polya process.

3. The method according to claim 1, wherein said non-stationary statistical processes comprise a process which employs Bose-Einstein statistics.

4. The method according to claim 1, wherein said at least one processing layer acts as a parser for written language input.

5. The method according to claim 1, wherein said at least one processing layer acts as a parser for oral language input.

6. The method according to claim 1, wherein said at least one processing layer acts as a datafile search program.

7. The method according to claim 1, wherein said at least one processing layer acts as a translator from one language to a second language.

8. The method according to claim 1, wherein said at least one processing layer acts to edit scanner inputs to produce corrected word processing files or printed text.

9. The method according to claim 1, wherein said at least one processing layer acts to calibrate oral input for speaker characteristics for input to subsequent processing by an oral parser.

10. A neural network system for processing information, comprising: a plurality of node means arrayed in a plurality of layer means to transform input arrays from prior layer means or the environment into output arrays with fractal dimension for subsequent layer means or output devices; means for applying learning rules based on non-stationary statistical processes; interconnection means between node means, the number and parameters of said interconnections being self-adjusted by said learning rule means during learning; at least one of said plurality of layer means acting as a processing layer means; a plurality of connection means from a data retrieval means to each node means of each of said at least one processing layer means.

11. The neural network system according to claim 10 wherein said non-stationary statistical process comprises a Polya process.

12. The neural network system according to claim 10, wherein said non-stationary statistical process comprises a process which employs Bose-Einstein statistics.

13. The neural network system according to claim 10, wherein said at least one processing layer means comprises means for parsing written language input.

14. The neural network system according to claim 10, wherein said at least one processing layer means comprises means for parsing oral language input.

15. The neural network system according to claim 10, wherein said at least one processing layer means comprises means for searching a datafile.

16. The neural network system according to claim 10, wherein said at least one processing layer means comprises means for translating from a first language to a second language.

17. The neural network system according to claim 10, wherein said at least one processing layer means comprises means for editing scanner inputs to produce corrected word processing files or printed text.

18. The neural network system according to claim 10, wherein said at least one processing layer means comprises means for calibrating oral input for speaker characteristics for input to subsequent processing means by an oral parser.

19. A method for processing information, comprising the steps of: using a neural network to process said information, said step of using a neural network further comprising: using a plurality of nodes arrayed in a plurality of layers to transform input arrays from prior layers or the environment into output arrays for subsequent layers or output devices; using interconnections between nodes, the number and structure of said interconnections being self-adjusted by the learning rules during learning; using at least one layer to combine and annotate data elements from previous layers, and to transmit augmented data arrays to the subsequent layer; using a plurality of lateral inputs to each node of each of the said at least one processing layer to retrieve information; applying learning rules which reinforce array elements that match data from templates retrieved from said plurality of lateral inputs, simultaneously suppressing array elements that do not match data from templates retrieved from said plurality of lateral inputs.

20. The method according to claim 19, wherein said learning rules comprise a Polya process.

21. The method according to claim 19, wherein said learning rules obey Bose-Einstein statistics.

22. The method according to claim 19, wherein said at least one processing layer acts as a parser for written language input.

23. The method according to claim 19, wherein said at least one processing layer acts as a parser for oral language input.

24. The method according to claim 19, wherein said at least one processing layer acts as a datafile search program.

25. The method according to claim 19, wherein said at least one processing layer acts as a translator from one language to a second language.

26. The method according to claim 19, wherein said at least one processing layer acts to edit scanner inputs to produce corrected word processing files or printed text.

27. The method according to claim 19, wherein said at least one processing layer acts to calibrate oral input for speaker characteristics for input to subsequent processing by an oral parser.

28. A neural network system for processing information, comprising: a plurality of node means arrayed in a plurality of layer means to transform input arrays from prior layer means or the environment into output array means for subsequent layer means or output devices; interconnections between node means, the number and structure of said interconnections being self-adjusted by learning rules during learning; at least one layer means for combining and annotating data elements from previous layer means, and to transmit augmented data arrays to the subsequent layer means; a plurality of lateral input means to each node means of each of the said at least one processing layer means to retrieve information; means for applying learning rules which reinforce array elements that match data from templates retrieved from said plurality of lateral input means, simultaneously suppressing array elements that do not match data from templates retrieved from said plurality of lateral input means.

29. The neural network system according to claim 28 wherein learning rules comprise a Polya process.

30. The neural network system according to claim 28, wherein said learning rules obey Bose-Einstein statistics.

31. The neural network system according to claim 28, wherein said at least one processing layer means comprises means for parsing written language input.

32. The neural network system according to claim 28, wherein said at least one

processing layer means comprises means for parsing oral language input.

33. The neural network system according to claim 28, wherein said at least one processing layer means comprises means for searching a datafile.

34. The neural network system according to claim 28, wherein said at least one processing layer means comprises means for translating from a first language to a second language.

35. The neural network system according to claim 28, wherein said at least one processing layer means comprises means for editing scanner inputs to produce corrected word processing files or printed text.

36. The neural network system according to claim 28, wherein said at least one processing layer means comprises means for calibrating oral input for speaker characteristics for input to subsequent processing means by an oral parser.